Optimizing Network Performance in Virtual Machines

THÈSE N^{O} 4267 (2009)

PRÉSENTÉE LE 27 JANVIER 2009

À LA FACULTE INFORMATIQUE ET COMMUNICATIONS

Laboratoire de systèmes d'exploitation

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Aravind MENON

acceptée sur proposition du jury:

Prof. A. Shokrollahi, président du jury
Prof. W. Zwaenepoel , directeur de thèse
Prof. A. L. Cox, rapporteur
Prof. B. Falsafi, rapporteur
Prof. S. Hand, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2009

Abstract

In recent years, there has been a rapid growth in the adoption of virtual machine technology in data centers and cluster environments. This trend towards server virtualization is driven by two main factors: the savings in hardware cost that can be achieved through the use of virtualization, and the increased flexibility in the management of hardware resources in a cluster environment. An important consequence of server virtualization is the negative impact it has on the networking performance of server applications running in virtual machines (VMs).

In this thesis, we address the problem of efficiently virtualizing the network interface in Type-II virtual machine monitors. In the Type-II architecture, the VMM relies on a special, 'host' operating system to provide the device drivers to access I/O devices, and executes the drivers within the host operating system. Using the Xen VMM as an example of this architecture, we identify fundamental performance bottlenecks in the network virtualization architecture of Type-II VMMs.

We show that locating the device drivers in a separate host VM is the primary reason for performance degradation in Type-II VMMs, because of two reasons: a) the switching between the guest and the host VM for device driver invocation, and, b) I/O virtualization operations required to transfer packets between the guest and the host address spaces. We present a detailed analysis of the virtualization overheads in the Type-II I/O architecture, and we present three solutions which explore the performance that can be achieved while performing network virtualization at three different levels: in the host OS, in the VMM, and in the NIC hardware.

Our first solution consists of a set of packet aggregation optimizations that explores the

performance achievable while retaining the Type-II I/O architecture in the Xen VMM. This solution retains the core functionality of I/O virtualization, including device driver execution, in the Xen 'driver domain'. With this set of optimizations, we achieve an improvement by a factor of two to four in the networking performance of Xen guest domains.

In our second solution, we move the task of I/O virtualization and device driver execution from the host OS to the Xen hypervisor. We propose a new I/O virtualization architecture, called the TwinDrivers framework, which combines the performance advantages of Type-I VMMs with the safety and software engineering benefits of Type-II VMMs. (In a Type-I VMM, the device driver executes directly in the hypervisor, and gives much better performance than a Type-II VMM). The TwinDrivers architecture results in another factor of two improvement in networking performance for Xen guest domains.

Finally, in our third solution, we describe a hardware based approach to network virtualization, in which we move the task of network virtualization into the network interface card (NIC). We develop a specialized network interface (CDNA) which allows guest operating systems running in VMs to directly access a private, virtual context on the NIC for network I/O, bypassing the host OS entirely. This approach also yields performance benefits similar to the TwinDrivers software-only approach.

Overall, our solutions help significantly bridge the gap between the network performance in a virtualized environment and a native environment, eventually achieving network performance in a virtual machine within 70% of the native performance.

Keywords: virtual machines, network virtualization, performance

Résumé :

Au cours des dernières années, les technologies de machines virtuelles dans les centres de données et les grappes de PC ont progressées de manière très rapide. Cette progression de la virtualisation dans le monde des serveurs est attribuable à deux facteurs principaux : les économies en matériel possibles grâce à la virtualisation, ainsi que la flexibilité de gestion des ressources virtualisées. Une conséquence importante de la virtualisation est limpact notable sur la performance des applications tournant sur les serveurs virtualisés.

Dans cette thèse, nous étudions les performances du réseau des machines virtuelles tournant dans lenvironnement de virtualisation Xen. Nous identifions les goulets détranglement fondamentaux qui limitent les performances du réseau dans la pile de virtualisation Xen, et nous proposons un certain nombre doptimisations afin daméliorer les performances des domaines Xen invités.

Nous présentons trois classes principales doptimisation. Pour commencer, nous proposons un ensemble doptimisations qui peuvent être appliquées sans modifier larchitecture actuelle de la du réseau dans Xen. Ces optimisations résultent en un facteur deux à quatre damélioration des performances réseau dans les domaines Xen invités

Ensuite, nous proposons une nouvelle architecture des entrées sorties (e/s) dans Xen appelée TwinDrivers, qui combine les avantages darchitectures e/s existantes. Plus spécifiquement, TwinDrivers combine la performance des VMMs basé sur un Hypervisor ainsi que la sécurité et la simplicité des VMMs basés sur un système hôte. Larchitecture TwinDrivers contribue un deuxième facteur deux damélioration dans la performance réseau des domaines Xen invités.

Pour terminer, nous décrivons une alternative matérielle à la virtualisation réseau, dans laquelle nous développons une interface réseau spécialisée (CDNA) qui permet laccès concurrent au réseau depuis plusieurs domaines Xen. Cette approche résulte en un gain de performance similaire à lapproche logicielle TwinDrivers.

Mots-clés : machines virtuelles, virtualisation du réseaux, performances

Acknowledgments

I would like to thank my advisor, Willy Zwaenepoel, for his patient support and encouragement over the course of my PhD, and his contributions to my thesis. Many of the papers that I published during my PhD would not have been possible without his final 'push', and his commitment to constantly improve the quality of the paper.

I would like to thank my committee members, Prof. Alan L. Cox, Prof. Steven Hand, and Prof. Babak Falsafi for reading this dissertation and providing constructive feedback. I would like to thank Prof. Amin Shokrollahi for serving as president of my jury.

Over the years, I had the good fortune to work with a number people who have helped define my thesis, and contributed ideas. I would like to thank Jose Renato Santos, Yoshio Turner and John Janakiraman from HP Labs for hosting me for a summer internship in 2004, and helping me get started on this thesis work. I would like to thank Prof. Alan Cox, Prof. Scott Rixner, Paul Willmann and Jeff Schafer from Rice University for giving me an opportunity to work with them on the CDNA project. I would like to thank Simon Schubert for working with me on the TwinDrivers project.

The members of the LABOS group, and the larger systems group, provided a wonderful work environment in which it was hard not to be productive, complete with serious academic sessions on comparative astrology and coffee breaks at the drop of a hat. My thanks to all members of the group: Oliver, Ming, Steve, Manu, Nikola, Simon, Rodrigo, Sameh, Katerina, Denisa, Nevena, Madeleine and Suzy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, there has been a rapid growth in the adoption of virtual machine technology in data centers and in cluster environments. Commercial virtualization software vendors such as VMware, XenSource (Citrix) and Microsoft are increasingly making their presence felt in the server market. On the hardware front, Intel and AMD have incorporated support for virtualization in their CPU instruction set [AMD, INT], and the PCI-SIG vendor consortium has proposed standards for adding support for virtualization in the PCI standard for I/O virtualization [ATS, SRI, MRI].

This trend towards server virtualization is driven by two main factors: the savings in hardware cost achieved through the use of virtualization, and the greater flexibility of management of virtualized cluster resources.

In a typical server environment, each server application is run on a separate server machine, both in order to isolate it from other server applications, and to customize its software stack (including the operating system). Unfortunately, this results in unnecessary server sprawl, where each server machine runs at a low average utilization rate. Server virtualization prevents this sprawl by consolidating multiple underutilized servers onto a smaller number of physical machines, thereby reducing the hardware costs required to run multiple servers. Each server application is run in its own operating system running on a different *virtual machine* (VM),

and multiple VMs are multiplexed on a single physical machine by the *virtual machine monitor* (VMM).

On the management front, cluster wide server virtualization allows for the decoupling of the management of the servers from the management of the hardware environment. Virtualization of cluster resources allows for increased flexibility in the centralized management of storage, ease of performing hardware upgrades and maintenance, and disaster recovery. For instance, virtual machine migration allows hardware upgrades to be performed with zero downtime by migrating the running servers onto other machines in the cluster while the target machine is being upgraded/fixed.

1.1 Virtualization Approaches

In a virtualized environment, the VMM performs the task of virtualizing and multiplexing the physical resources of the system among the virtual machines. These resources include the CPU, memory and I/O devices. Thus, it provides each VM with its own virtual CPU(s), a subset of the host memory, and a set of virtual I/O devices. The guest OS running in the VM uses these virtual resources in a manner similar to the way it uses physical resources in a native environment.

There are two main distinct approaches to virtualization, full virtualization and paravirtualization [BDF+03, WSG02]. In full virtualization, the hardware interface provided to guest operating systems is exactly identical to the underlying native hardware interface. The advantage of this approach is that it provides full binary compatibility with existing operating systems, and thus does not require any modification in the guest OS to run in the VM. The downside of this approach is performance degradation incurred for emulating the exact semantics of the native hardware. The full virtualization approach for x86 CPUs was pioneered by VMware. With the introduction of hardware support for virtualization in x86 CPUs, other virtualization products, such as Citrix's XenServer, and Microsoft Hyper-V also support full virtualization for unmodified OSes.

Because of the performance overheads of full virtualization, and also in order to avoid the complexity of virtualizing x86 CPUs in the absence of hardware support, an alternate approach to virtualization called paravirtualization was developed. In this approach, which was first developed in the Xen VMM, the guest operating system running in a VM is slightly modified to make it aware of the underlying VMM, so that, instead of accessing the raw hardware directly, it requests the VMM to perform privileged operations on the hardware on its behalf. The advantage of paravirtualization is improved performance over full virtualization, whereas the downside is the need to modify guest operating systems in order to run in a VM.

1.2 Network Performance in Virtual Machines

In a virtualized environment, the VMM multiplexes I/O devices among the VMs by providing each VM with a set of virtual I/O devices. In the case of networking, each VM is provided with one or more virtual network interfaces. The guest OS uses this virtual network interface for all its network I/O in a manner similar to the way it uses a regular network interface in a native environment.

The VMM is responsible for multiplexing the network I/O operations performed by the guest OSes on the virtual network interfaces onto the physical network interface. Thus, for transmit operations, it must transfer packets transmitted on the guest's virtual interface onto the physical network interface card (NIC) (or to another guest VM, if the target of the packet runs on a VM on the same machine). For receive operations, it must route packets in the reverse direction, i.e., from the physical NIC to the appropriate virtual interface in a VM. This packet transfer operation between the virtual and physical NICs is typically done through an L-2 bridge and constitutes the main function of the network virtualization stack.

The complexity and cost of network virtualization in the VMM depends on a number of factors, with an important factor being the I/O architecture used by the VMM. The I/O architecture of the VMM defines the way it uses device drivers to control the I/O devices in the system. There are two main approaches to do this, and VMMs are classified as either Type-I

or Type-II depending on which approach they take.

In a Type-I VMM, or non-hosted VMM (also called hypervisor-based VMM), the virtual machine monitor completely controls all hardware resources in the system, including the I/O devices, and no guest OS is allowed to access the hardware directly. Thus, the VMM provides its own device drivers for controlling the I/O devices in the system. The main advantage of this approach is that I/O operations in the virtualized environment are very efficient, because the hypervisor can directly invoke its drivers for performing I/O. The downside of this approach is the software engineering effort required to develop or port the device drivers and their driver support library. A second disadvantage is that the hypervisor becomes vulnerable to bugs in the driver, and an I/O operation on behalf of one VM can bring down the entire system.

In a Type-II VMM, also called a hosted VMM, the VMM does not control all hardware directly. Instead, it relies on a special privileged *host* operating system (called *driver domain* in Xen terminology) for controlling and managing the hardware, including all I/O devices. Thus, device drivers for managing the I/O devices are provided by the host OS, and the VMM must switch from the guest to the host OS for every device I/O operation. Thus, this approach entails greater performance overhead for I/O operations. The advantage of this approach is that, firstly, the VMM can simply reuse the device driver and its support infrastructure present in the host OS, saving on considerable engineering effort, and, secondly, since the device driver runs in an isolated host VM, bugs or crashes in the driver cannot affect the rest of the system.

1.3 Scope of Thesis

An important concern with server virtualization is that the performance of server applications running inside the VMs can degrade significantly relative to their performance in a native environment. This is because of the overheads incurred by the VMM in the virtualization of the physical resources of the system. This overhead can be particularly high for the virtualization of I/O devices [MST+05, SVL01], especially for I/O devices which require frequent servicing by the device driver, for example, interrupt-intensive devices such as network cards.

We address the problem of efficiently virtualizing the network interface in a virtual machine environment. A number of important server applications, such as web servers, and streaming media servers are network-intensive applications, and their overall performance depends crucially on the networking performance of the system. In addition, efficient networking in the VMM is useful for a number of VMM operations, such as virtual machine migration, network file system traffic, etc.

In this thesis, we investigate the networking performance in a Type-II VMM, using the Xen VMM as an example of this architecture. Xen is an open-source VMM developed at the University of Cambridge [BDF+03], and it uses an I/O architecture similar to the Type-II hosted VMM. Although the Xen VMM controls all physical hardware other than the I/O devices directly (and in this respect, is similar to a Type-I VMM), it requires a special, privileged host OS, called the driver domain, to control and manage the I/O devices. Guest operating systems running in other VMs (known as *guest domains* in Xen terminology) must request I/O services from the driver domain with the help of Xen-specific paravirtualized drivers. Since Xen uses a hosted I/O architecture, it suffers from the switching and other virtualization overheads associated with the Type-II VMM architecture for I/O operations.

We focus on the network performance in paravirtualized guest operating systems running on Xen. We identify the fundamental performance bottlenecks in the network virtualization stack of the Xen VMM, and propose a number of solutions to address these bottlenecks.

1.4 Contributions of this Thesis

We explore a number of complementary solutions to optimize network virtualization in Type-II VMMs. Although our solutions have been implemented and evaluated in the Xen VMM, the techniques we propose are general, and can be applied to other Type-II VMMs, such as Microsoft's Hyper-V.

The contributions of this thesis are the following:

Using the Xen VMM as an example of Type-II VMMs, we identify the fundamental per-

formance bottlenecks in the network virtualization architecture of Type-II VMMs. We show that locating the device drivers in a separate host VM is the primary reason for performance degradation in Type-II VMMs, because of two reasons: 1) the switching between the guest and the host VM for device driver invocation, and, b) the overhead of I/O virtualization operations required to transfer packets between the guest and host address spaces. We show that in the Xen VMM, these overheads degrade network performance by up to a factor of five relative to the native performance.

We present three solutions which explore the space of networking optimizations in Type-II VMMs. The three solutions explore the networking performance that can be achieved through different optimizations while performing network virtualization at different levels: in the driver domain (host OS), in the VMM, and in the NIC hardware.

Our first solution explores the networking performance that is achievable while retaining the Type-II I/O architecture of Xen. We present a set of optimizations that reduce network virtualization overheads in the I/O stack by using packet 'coalescing' techniques to reduce the per-packet processing overheads. This solution retains the core functionality of I/O virtualization, including device driver execution, in the Xen driver domain. These optimizations improve TCP transmit performance in Xen guest domains by a factor of 4, and TCP receive performance by a factor of roughly 2, relative to the baseline performance.

In our second solution, we move the task of I/O virtualization and device driver execution from the host OS to the Xen hypervisor. We propose the TwinDrivers architecture that allows us to automatically *derive* safe network drivers from the host OS drivers, and run these drivers directly in the hypervisor, like in a Type-I VMM. The TwinDrivers architecture combines the performance advantages of Type-I VMMs with the safety and software engineering benefits of Type-II VMMs. We improve networking performance in Xen guest domains by a factor of 2 using TwinDrivers.

Our third solution consists of a hardware-based approach to improve networking performance. This work was done in collaboration with researchers at Rice University [WSC+07]. In

this approach, we move the task of network virtualization into the network interface card (NIC). We develop a specialized network interface (Concurrent Direct Network Access, CDNA) which allows guest domains running in VMs to directly access a private, virtual context on the NIC for network I/O, completely bypassing the host OS. This approach also helps achieve network performance comparable to Type-I VMMs.

Overall, our solutions help significantly bridge the gap between the network performance in a virtualized environment and native environment, eventually achieving within 70% of the native performance.

The outline of the rest of the thesis is as follows. In Chapter 2, we present an analysis of the network virtualization overheads in Type-II VMMs, with an emphasis on the Xen VMM. In Chapter 3, we describe our packt aggregation optimizations that retain the Xen I/O architecture. Chapter 4 describes our TwinDrivers approach to I/O virtualization, and Chapter 5 describes our hardware based CDNA approach. We discuss related work in Chapter 6, and we conclude in Chapter 7.

Chapter 2

Network Virtualization Overheads

In this chapter, we describe the network I/O virtualization architecture of Xen, and analyze the performance characteristics of the Xen I/O architecture (and Type-II VMM architectures in general), for network-intensive workloads.

We first present the network I/O architecture used in Xen. The Xen VMM uses an I/O architecture similar to that of hosted Type-II VMMs. Although the Xen hypervisor controls most of the physical hardware directly (such as CPU and memory), it delegates the control of the I/O devices to a privileged host OS running inside a VM, called the driver domain. Thus, the performance characteristics of the Xen VMM for I/O workloads are similar to that of Type-II VMMs. We describe the Xen I/O architecture and the implications of this architecture on network performance in Section 2.1. Next, we analyze the main sources of performance overhead in the Xen I/O virtualization architecture for network intensive applications in Section 2.2. The section presents a high level picture of the fundamental performance bottlenecks in the Type-II I/O architecture of Xen. Finally, we present a more detailed analysis of the network processing overheads incurred for transmit and receive workloads running in Xen guest domains in Section 2.3. This section identifies the components of the I/O virtualization stack in Xen that dominate network processing overhead.

2.1 Xen I/O Architecture

Figure 2.1 shows the architecture of a Type-I VMM, also known as a hypervisor-based VMM, or non-hosted VMM. In this architecture, the VMM controls all hardware resources of the system, including the I/O devices. Thus, the VMM provides device drivers of its own to manage the I/O devices. In contrast, in a Type-II architecture, the VMM relies on a privileged host operating system to manage all the hardware, in particular, the I/O devices. Figure 2.2 shows the architecture of a Type-II VMM.

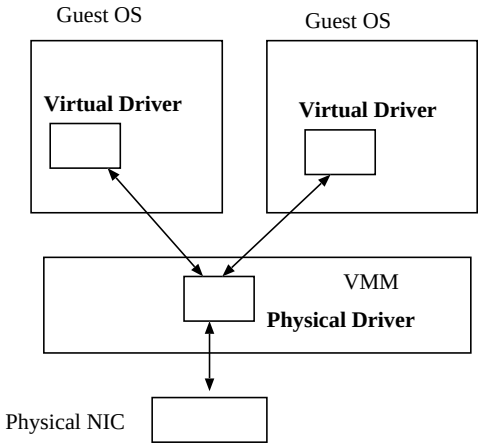

Figure 2.1: Type-I VMM Architecture. Device drivers for controlling the I/O devices are present in the hypervisor (VMM).

The advantage of the Type-II architecture is that the VMM can reuse device drivers from the host OS to manage a wide range of I/O devices. This constitutes a significant saving in the software engineering effort required to make the VMM portable across a range of hardware platforms. Unfortunately, this architecture also suffers from higher performance overhead compared to Type-I VMMs, especially for I/O intensive operations. This is because for every invocation of the device driver, the VMM must switch from the currently running guest OS to the host OS hosting the drivers.

The VMM architecture of Xen is a combination of Type-I and Type-II VMM. Although the Xen hypervisor controls the physical hardware such as CPU and memory directly, it delegates

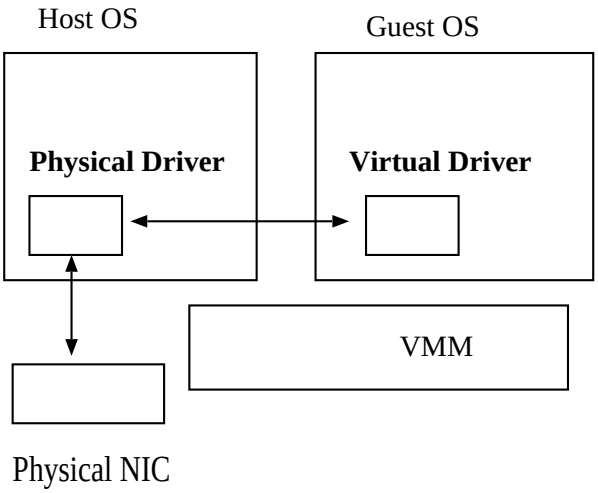

Figure 2.2: Type-II VMM Architecture. The VMM makes use of the device drivers of the host OS for accessing I/O devices.

control of I/O devices to a special, privileged operating system running in a VM, called the driver domain. Thus, it is similar to a hosted VMM architecture in that it relies on a host OS for managing the I/O devices. However, Xen runs the host OS in an isolated VM of its own, thus, the VMM and other VMs are isolated from bugs or crashes causes by the device driver.

Figure 2.3 shows a high level picture of the network I/O virtualization architecture of Xen. Each guest domain running on Xen is provided with a number of virtual network interfaces which it uses for all its network operations. Each virtual interface in a guest domain (also called frontend interface) has its own MAC address and is connected to a corresponding 'backend' interface in the driver domain through an 'I/O channel'. The I/O channel provides mechanisms for exchanging network packets between the frontend and backend interfaces. The frontend and backend interface can signal each other using virtual interrupts when there are network packets to be transferred between the two.

All the backend interfaces in the driver domain (corresponding to the guest domain's virtual interfaces) are connected to the physical NIC and to each other through a network bridge. The combination of the bridge and the I/O channel allows the physical interface and the guest domain's virtual interfaces to transfer packets to each other based on the destination MAC

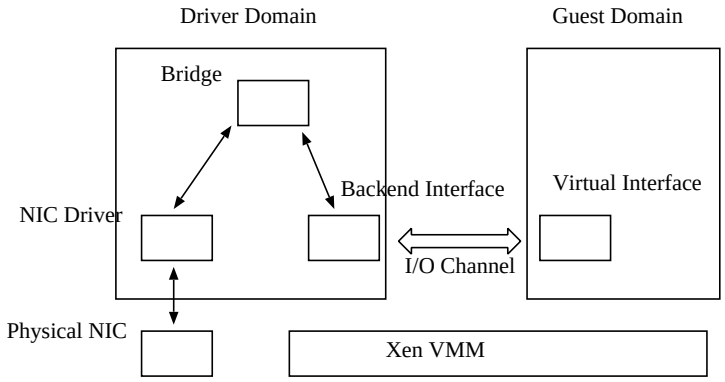

Figure 2.3: Xen Network I/O Architecture

address of the packet. Thus, for instance, on the transmit path, packets are transmitted by the guest domain on its virtual interface, which are then transferred over the I/O channel to the backend interface, which then transfers them to the physical interface over the network bridge, and finally the NIC sends them out on the network. The receive path is similar, except in the reverse direction.

The network virtualization overheads in the Xen I/O architecture are similar to the I/O overheads incurred in a Type-II VMM. Like in a Type-II VMM, network I/O operations in the guest domain require an address space switch to the driver domain in order to invoke the NIC driver. The frequent switching for I/O operations and the cost of operations in the I/O virtualization stack (I/O channel transfers, bridging) significantly degrade network performance in Xen guest domains, as shown in the next section.

2.2 Xen Virtualization Overheads

Figure 2.4 shows the transmit and receive network performance for a netperf [NET] like benchmark running in three configurations: a Xen guest domain, the Xen driver domain, and a native Linux system. The benchmark measures the maximum transmit/receive throughput achievable (in Mb/s) over a small number of TCP connections, where each TCP connection is used to send/receive traffic over a different network card. The first set of histograms in the figure shows

the transmit performance and the second set shows the receive performance.

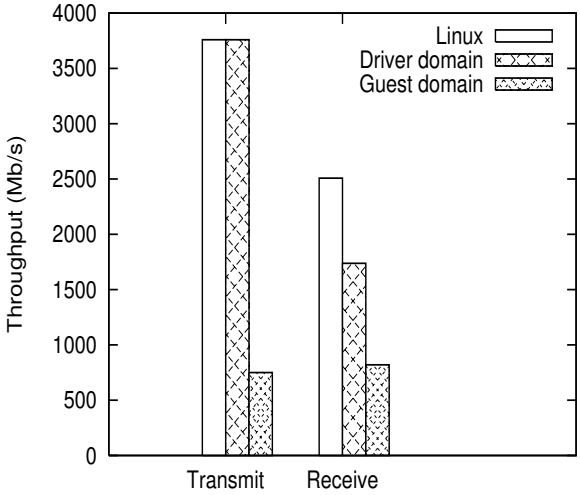

Figure 2.4: Network performance in guest and driver domains running on Xen

The network performance when running the workloads in the driver domain is very close to the performance in a native Linux system. Thus, transmit performance (3760 Mb/s) is the same as the Linux transmit performance (except that it incurs higher CPU overhead), and receive performance (1738 Mb/s) is within 70% of the native receive performance (2508 Mb/s). In contrast, when the workloads run in the guest domain, the performance achieved is significantly lower. The transmit performance in the guest domain (750 Mb/s) is only 20% of the native transmit performance, and the receive performance (820 Mb/s) is roughly 33% of native.

The reason for the large difference between guest domain and driver domain performance is that the driver domain invokes the NIC driver directly for network I/O, whereas the guest domain needs to switch to the driver domain, and transfer its packet data to the driver domain address space, for all network I/O operations. We note that in a Type-I VMM, no switching is incurred for invoking the device driver in the hypervisor from the guest domain, and the packet remains in the same address space, thus the performance of a Type-I VMM is comparable to the performance of the Xen driver domain.

We now coarsely quantify the virtualization overheads of the Type-II architecture for running

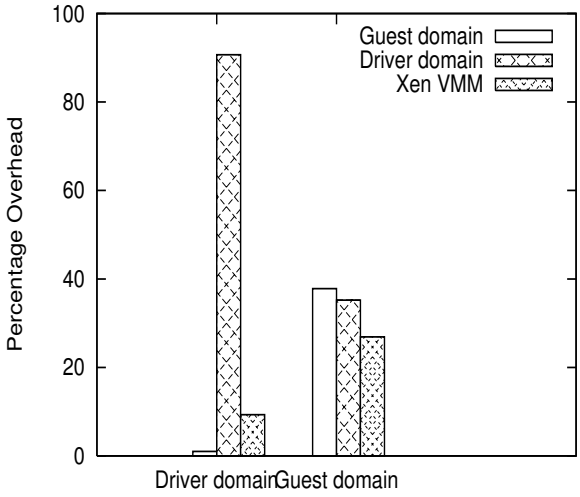

Figure 2.5: Breakdown of network processing overheads for guest and driver domains running on Xen

network-intensive workloads in the guest domain. Figure 2.5 shows the breakdown of network processing overhead for the transmit workload for guest and driver domain configurations. The figure shows the percentage of the total network processing overhead that is incurred in the driver domain, the Xen VMM, and the guest domain.

When the network workload is run in the driver domain (shown in the first set of bars), the I/O virtualization overhead incurred in the hypervisor is less than 10%, and most of the processing time is spent in the driver domain itself. However, when the workload runs in the guest domain (shown in the second set of bars), close to 62% of the processing time is spent in the network virtualization stack in the driver domain (35%) and the Xen hypervisor (27%), and only 38% of processing time is left for network processing in the guest domain. The processing overhead in the driver domain (35%) and the Xen hypervisor (27%) on behalf of the guest domain network I/O, is composed almost entirely of I/O virtualization operations used to transfer packets from the guest address space (virtual NIC) to the driver domain address space (physical NIC) (We discuss this in more detail in Section 2.3).

We quantify the impact of the context switching between the driver and the guest domain on the performance of the network stack. Figure 2.6 shows the relative ratio of the number of

hardware events such as L-2 cache misses, instruction and data TLB misses, for the transmit workload running in the guest and driver domain configurations. The figure shows the number of misses in each configuration as a fraction of the number of misses incurred in the guest domain. The figure shows that network I/O from the guest domain incurs significantly higher overhead, in terms of cache and TLB misses, than the driver domain. L-2 cache misses incurred are higher in the guest domain by a factor of almost 25, instruction TLB misses are higher by a factor of 2.5 and data TLB misses are higher by a factor of 3. These figures indicate that the constant switching between the guest and the driver domain for network I/O significantly degrades the performance of the operations in the network I/O stack.

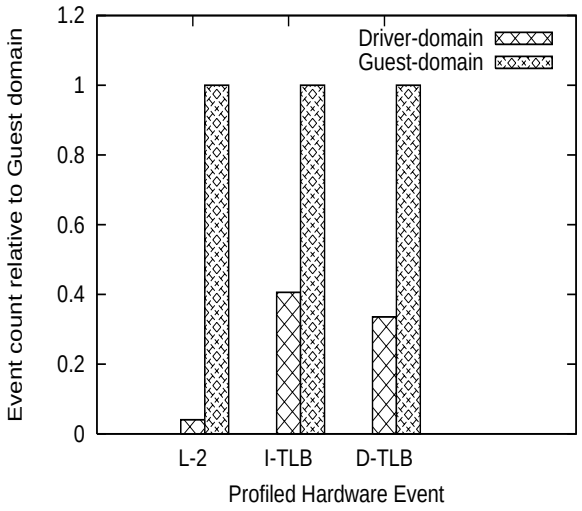

Figure 2.6: Hardware event counts for network workload running in Xen guest and driver domains

Figures 2.5 and 2.6 show that the choice of the driver domain I/O architecture for network I/O is the fundamental bottleneck for network performance in Xen guest domains. Significant overhead is incurred in the I/O virtualization stack in transferring packets between guest domain and the driver domain address spaces, and this is exacerbated by the overhead of switching between the guest and driver domains. In contrast, in the driver domain configuration, the network driver can be invoked directly, and this shows an estimate of the performance that is achievable in a Type-I VMM architecture. We now discuss the I/O virtualization overheads of

the Xen architecture in more detail in the next section.

2.3 Analysis of Xen Virtualization Overheads

We identify the components of the network I/O virtualization stack that contribute the most to virtualization overhead for guest domains. An important objective here is to quantify the overhead of operations in the network stack that must be necessarily incurred on each network packet, such as TCP/IP protocol processing, and distinguish it from other operations that are not necessarily per-packet, such as the I/O virtualization specific operations.

In the following analysis, we also distinguish between 'per-packet' overheads in the network stack, and 'per-byte' overheads. In the networking literature, per-packet overhead is defined as the overhead incurred in operations that must be performed for every *packet* of data processed, and includes operations such as packet protocol header processing, buffer management, I/O channel transfers, etc. In contrast, the per-byte operations are those which must be performed on every *byte* of data processed, and includes operations such as data copying and checksum computation.

Figures 2.7 and 2.8 show the breakdown of the processing overhead for a transmit and a receive workload, respectively, running in a Linux 2.6.16.38 guest domain running on Xen 3.0.4. The figures show the overhead incurred for netperf [NET] like transmit and receive workloads in different components of the network virtualization stack of Xen, in terms of number of CPU cycles incurred per network packet. The different groups into which we classify the network processing overheads are the following:

1. **per-byte**: Data copy routines. For the transmit workload this includes the data copy from the guest domain's file system buffers into the frontend driver's socket buffers. For the receive workload, this includes the two data copies: the first from the driver domain into the guest domain, and second copy from the guest kernel into the guest application.

2. **non-proto**: This includes the bridge and netfilter routines in the driver domain, and

other system-specific non-protocol processing related routines in the guest domain. The bridge transfers packets from the backend interface to the physical NIC for transmit workloads, and in the reverse direction for receive workloads. The overhead in this category is essentially a per-packet overhead.

3. **netback**: The netback driver initializes transfer of ACK packets from the driver to the guest domain for transmit workloads, and receives packets from the guest domain for receive workloads. Its overhead is mostly per-packet, and is proportional to the number of packets it transfers.

4. **netfront**: The netfront driver initiates transfer of packets from the guest domain to the driver domain for transmit workloads, and receives packets from the driver domain for receive workloads. Its overhead is similar to the netback driver, and is proportional to the number of packets it accepts.

5. **tcp rx and tx**: The transmit and receive TCP routines in the guest domain. These are per-packet overheads.

6. **buffer**: Buffer management routines for management of network packet buffers and the Linux `sk_buff` structure, in both the driver domain and the guest domain. This is a per-packet overhead.

7. **driver**: Device driver running in the Driver domain. This is a per-packet overhead.

8. **xen**: Xen manages domain scheduling, inter-domain interrupts, validation of packet transfer rights, etc. Its overhead cannot be classified strictly as either per-packet or per-byte.

9. **misc**: Other routines. These cannot be classified as either per-packet or per-byte.

For transmit processing (figure 2.7), the overhead of the device driver routines, `driver`, is roughly 11%, and for receive side processing (figure 2.8), it is roughly 8%. Although this

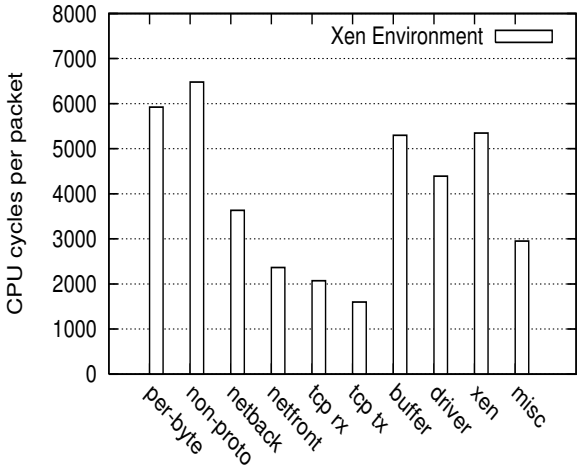

Figure 2.7: Breakdown of transmit processing overheads in a Linux guest domain running on the Xen VMM. The overhead is predominantly per-packet, and most of the per-packet overhead is incurred in virtualization-specific operations such as bridging, etc. TCP/IP protocol processing incurs very low overhead.

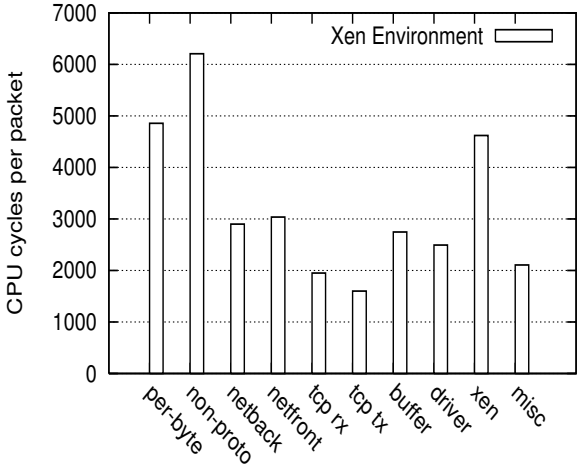

Figure 2.8: Breakdown of receive processing overheads in a Linux guest domain running on the Xen VMM. The overhead is predominantly per-packet, and most of the per-packet overhead is incurred in virtualization-specific operations such as bridging, etc. TCP/IP protocol processing incurs very low overhead.

overhead is per-packet, it cannot be reduced by software-only optimizations, and possibly requires modifications to the NIC for further reduction. Thus, for the purpose of the subsequent

discussion, we distinguish between the `driver` routines and the other per-packet routines in the network stack.

The overall overhead of the per-packet routines for the transmit workload, comprising of the `non-proto`, `netback`, `netfront`, `tcp tx`, `tcp rx` and `buffer` routines, adds up to roughly 50% of the total network processing overhead, and is significantly larger than the per-byte overhead, 14%. Of this total per-packet overhead, TCP/IP protocol processing, consisting of `tcp tx` and `tcp rx`, is only 7% of the network processing overhead. The rest of the per-packet overhead is incurred in operations unrelated to protocol processing, such as L-2 bridging in the driver domain (`non-proto`, 15%), I/O channel transfers (`netfront` and `netback`, 14%), and Linux buffer management (`buffer`, 12%). The overhead of these operations is possibly further exacerbated because of the switching overheads described in section 2.2.

The overhead of the bridging operations in the driver domain (the `non-proto` category) is 15%. One of the reasons for this being so large is that the L-2 bridge in the driver domain includes the infrastructure for the Linux netfilter framework, which provides a set of hooks into the kernel for intercepting and manipulating network packets. Unfortunately, the overhead of this infrastructure is non-trivial even when no netfilter hooks are set up.

The second major source of overhead is the I/O channel operations (`netback` and `netfront`, 14%) for transferring packets between the guest and driver domains. This operation is expensive because, for every packet transferred between the driver and the guest domain, the driver domain must verify the permissions of the guest domain to access the physical page address of the packet. In the Xen architecture, this operation requires hypercalls to the hypervisor in order to perform the so-called grant-table operations [FHN+04]. This overhead is specific to the Type-II VMM architecture, where the driver domain needs to invoke the hypervisor in order to verify the guest domain's permissions. In a Type-I architecture, the VMM can directly verify the guest domain's page permissions, since the device driver is invoked inside the hypervisor.

The profile for receive side workloads, in figure 2.8, is very similar to that of the transmit workload. The overall overhead of the per-packet routines for the receive workload adds up to

roughly 56% of the total overhead, and is significantly higher than the per-byte copy overhead, 14%. This is inspite of the fact that there are two data copies involved, one from the driver domain to the guest domain, and the second from the guest kernel to the guest application. TCP/IP protocol processing (`tcp tx` and `tcp rx`) incurs less than 10% of the overhead, and the non-protocol processing related per-packet operations, `non-proto`, `netback`, `netfront` and `buffer`, add up to almost 46% of the overhead. Here again, the bulk of the per-packet overhead is incurred in the I/O virtualization operations such as bridging (`non-proto`, 19%), and I/O channel operations (`netback` and `netfront`, 18%).

Thus, for both transmit and receive workloads, most of the per-packet overhead is incurred in routines in the I/O virtualization stack, and very little overhead in incurred in TCP/IP protocol processing.

2.4 Summary

In this chapter, we analyzed the performance characteristics of network-intensive workloads running in Xen guest domains. We showed that the I/O virtualization architecture of Xen, which is similar to that of Type-II VMMs, results in a significant performance degradation for I/O intensive workloads. In contrast, an I/O architecture similar to Type-I VMMs (such as the driver domain) suffers from much less performance degradation. We further showed that much of the overhead of network processing in guest domains is incurred in the I/O virtualization stack, in operations such as bridging and I/O channel transfers. Actual TCP/IP protocol processing itself incurs very little overhead.

In the next three chapters, we present three complementary solutions to reduce the I/O virtualization overheads in Type-II VMMs. The solutions proposed are motivated in part by the analysis and observations made in this chapter.

In chapter 3, we present a set of optimizations that reduce the per-packet I/O virtualization overheads in the existing Xen network stack. The optimizations in this chapter make use of our observation that a number of overheads in the I/O virtualization stack (such as L-2 bridging

and I/O channel transfers) are currently incurred on a per-packet basis, but since they are not central to TCP/IP protocol processing, they do not actually need to be incurred for every packet transmitted or received from the network.

Our second and third solutions try to eliminate the driver domain (or host OS) from the performance critical path of network I/O operations in a Type-II VMM, and thus, try to achieve performance closer to a Type-I VMM architecture. The two solutions take different approaches to achieve this goal.

In our second solution, presented in chapter 4, we propose the TwinDrivers I/O architecture, which allows us to automatically *derive* safe hypervisor drivers from the host OS drivers, and run these drivers directly in the hypervisor, like in a Type-I VMM. The rewriting is done in a manner that guarantees memory safety of the derived hypervisor driver. Thus, the TwinDrivers architecture combines the performance advantages of a Type-I VMM with the safety and software engineering benefits of a Type-II VMM.

In our third solution, in chapter 5, we eliminate the host OS from the performance critical path by using a hardware-based approach, called CDNA. In the CDNA approach, we move the task of network virtualization into the network interface card (NIC). We develop a specialized network interface that can be partitioned safely into independent virtual network contexts, such that each context can be assigned independently to a different guest domain. By performing network virtualization directly in the network interface, this approach bypasses the driver domain, and thus achieves performance similar to a Type-I VMM.

Chapter 3

Packet Aggregation Optimizations

In this chapter we present a set of optimizations to the Xen network stack that can be applied to the existing Xen I/O architecture. These optimizations retain the basic Xen architecture of locating device drivers in a privileged driver domain, and providing guest domains with paravirtual network interfaces for network I/O operations (see chapter 2). The optimizations address both transmit side and receive side networking performance.

The basic idea of our optimizations is to focus on reducing the per-packet overhead of packet processing in the network stack, as a means to reduce the overall network processing overhead. The optimizations reduce per-packet overheads in the network stack by batching together, or combining, the processing of multiple small network packets into one 'coalesced' host packet.

Batching together the processing of network packets is an effective way to reduce the per-packet overheads. This is because batching, or coalescing, allows many of the packet processing operations in the network stack to be performed only once for every 'coalesced' packet, instead of once for each network packet. Thus, for instance, operations such as bridging, I/O channel transfers and system specific operations such as buffer management (which are not related to TCP/IP protocol processing) now need to be performed only once for every *coalesced* packet, instead of being performed for every *network* packet. Thus, the average cost of these per-packet operations can be reduced to a small fraction of their original cost, by varying the 'batching

ratio'.

We present tranmit side and receive side optimizations that make use of the general philosophy of coalescing packets to reduce per-packet overhead. On the transmit side, we present two optimizations, a High Level Virtual Interface (HLVI) optimization for the guest domain's virtual network interface, and I/O channel optimizations. On the receive side, we also present two optimizations, Receive Aggregation and Acknowledgment Offload. The transmit side optimizations improve TCP transmit performance in guest domains by a factor of upto four, and the receive side optimizations improve receive performance by 86%.

The outline for the rest of this chapter is as follows. We describe the transmit side optimizations in section 3.1 and the receive side optimizations in section 3.2. We evaluate the performance benefits of these optimizations in section 3.3 We conclude in section 3.4 with a discussion of the limitations of these optimizations.

3.1 Transmit Side Optimizations

We present the High Level Virtual Interface optimization in section 3.1.1, and the I/O Channel Optimizations in section 3.1.2.

3.1.1 High Level Virtual Interface

A number of transmit side optimizations in the network stack of an operating system depend on hardware offload support from the network interface (NIC). For instance, support for TCP Segmentation Offload (TSO) is most commonly implemented in the network interface, and zero-copy TCP transmit implementation requires support for scatter/gather DMA and checksum offloading from the network interface. The TSO optimization allows the network stack to transmit packets using an effective MTU size which is much larger than the network MTU size. Scatter/gather DMA support allows the network card to directly DMA network packets from non-contiguous memory regions, such as from the OS file system buffers and a distinct protocol header.

Unfortunately, when a physical network interface is virtualized by a VMM, the corresponding virtual network interface provided to guest domains may not always support these offload features. This was, in fact, the case with the initial implementation of virtual network interfaces for Xen guest domains. Xen guest domains were initially provided with a basic, simple network interface which supported only the basic transmit and receive operations, and did not support any additional hardware offload features, even if these features were supported by the underlying physical NIC.

The rationale behind this choice of a low-level virtual network interface was that it simplified the implementation of the virtual NIC, and made it 'portable' across all underlying physical NICs. Since the virtual NIC supported only the least common denominator of all possible physical NIC functionality, its operations could be mapped easily to corresponding operations in the physical NIC. Unfortunately, this approach also prevented the Xen guest domain from taking advantage of the offload features of the physical NIC.

Our first optimization is a new 'High Level Virtual Interface' for Xen guest domains, which aims to reduce the per-packet overheads in the TCP transmit path for Xen guest domains. Figure 3.1 shows a high level picture of this virtual interface architecture.

In this architecture, we provide a 'high-level' virtual interface for networking in Xen guest domains, that supports all the important transmit side offload features such as TCP Segmentation Offload (TSO), scatter/gather DMA and checksum offload. The key feature of the new architecture is that the offload features are supported in the guest's virtual interface independent of whether they are supported in the underlying physical NIC.

Since the physical NIC may not support all the offload features required by the guest's virtual NIC, we introduce a new component, the 'offload driver', to emulate the required features in software. The offload driver is situated at the entry point of the NIC driver in the driver domain, and its task is to ensure that packets transmitted from the virtual interface are 'compatible' with the offload features supported in the physical NIC. Thus, if a packet from a guest domain requires TSO support in the NIC, and the physical NIC does not support this feature, the

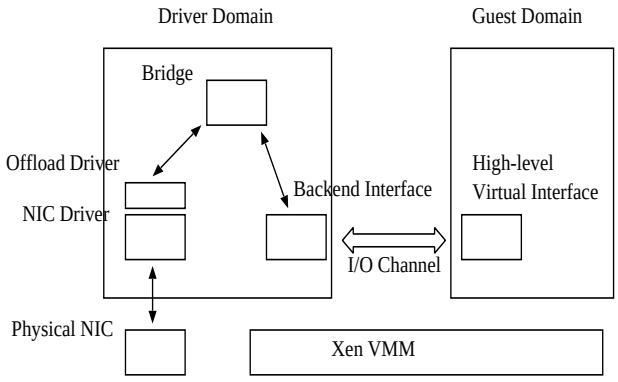

Figure 3.1: High Level Virtual Interface Architecture. The virtual NIC in the guest domain supports high level offload operations independent of whether they are supported by the actual NIC.

offload driver emulates TSO in software, and sends only segmented MTU-sized packets to the physical NIC. Similarly, the offload driver implements support for other offload features, like scatter/gather DMA and checksum offloading.

Advantages of a High Level Virtual Interface

The main advantages of a high level virtual interface are similar to the advantages of traditional transmit side optimizations, such as TSO and zero-copy transmit, in a native operating system network stack. Thus, scatter/gather DMA and checksum offload support in the virtual NIC allow the guest domain to perform zero-copy TCP transmits, and TSO support allows the guest operating system to make use of the standard large packet offload optimization.

However, the impact of these optimizations, in particular the TSO optimization, is even more pronounced in a virtualized environment than in a native environment. This is because, in a virtualized environment such as Xen, a large part of the network processing overhead is incurred in the network virtualization operations performed in the driver domain and the Xen VMM, and much of this overhead is incurred in per-packet operations not related to protocol processing, such as bridging and I/O channel transfers (see chapter 2). The TSO optimization amortizes the cost of these per-packet operations by batching together the processing of multiple

MTU-sized network packets into one 'large' host packet. Thus, operations in the virtualization stack are incurred only once for each 'large' packet, instead of once for each network packet.

This also justifies the decision to support the high level offload features in the virtual NIC, even when the underlying physical NIC does not support them. By supporting the offload features in the virtual NIC, in particular TSO, the cost of the per-packet operations in the network virtualization stack and in the guest domain's network stack are greatly reduced, even in the case when the features have to be emulated by the offload driver. Since the actual offload emulation takes place at a point just before invoking the physical NIC driver, few additional per-packet overheads are incurred after this point.

3.1.2 I/O Channel Optimizations

Our second transmit side optimization in the Xen architecture consists of I/O channel optimizations for improving the efficiency of packet transfer over the I/O channel. The I/O channel, as discussed in chapter 2, is the inter-domain data transfer mechanism used for exchanging packets between the frontend driver in the guest domain and the backend driver in the driver domain.

As such, this I/O channel optimization does not strictly fall in the general category of the packet coalescing optimization techniques discussed in this chapter. However, we include it for completeness to cover the set of transmit side optimizations which can be implemented within framework of the Xen I/O architecture. The description of these optimizations in this section is kept brief, additional details can be found in [MCZ06].

The I/O channel optimizations consists of a set of techniques designed to avoid the inefficiencies in the packet transfer mechanism used in Xen, for the common case. It consists of a transmit path optimization which avoids the cost of the I/O channel page remapping operations using a 'header' channel, and an ACK receive path optimization which replaces page remapping by data copy (We consider the receive path operations for the I/O channel for reception of the ACK packets in TCP).

On the transmit path (i.e., for transferring data from the frontend driver to the backend

driver), the I/O channel currently makes use of a page remapping mechanism in which a virtual memory page in the driver domain is mapped to the physical address of the data page in the guest domain. This is an expensive operation involving multiple TLB flushes and hypercalls (to check for page permissions). Our transmit path optimization relies on the observation that the driver domain does not need to map in the entire packet for its bridging operation. It only needs to access the MAC header of the packet. Thus, the optimization consists of augmenting the I/O channel with a 'header' channel, which the frontend driver in the guest domain uses to to supply the packet header separately to the backend driver. The header channel, thus, avoids the page remapping operation and replaces it with the cost of a small header copy.

Similarly, on the receive path (transferring ACK packets from the driver domain to the guest domain), the initial I/O channel mechanism made use of complex page remapping and page ownership transfer operations in order to avoid data copy costs. However, for TCP transmit workloads, packets transferred on the receive path are only small TCP ACK packets, whose data copy costs are quite small. Thus, our receive path optimization consists of using data copy to implement the I/O channel transfer from driver to guest domain.

3.2 Receive Side Optimizations

We now present receive side optimizations which can be applied within the framework of the Xen I/O architecture. We discuss the Receive Aggregation optimization in section 3.2.1 and the Acknowledgment Offload optimization in section 3.2.2.

Both these optimizations fall in the general category of packet coalescing optimizations discussed earlier. The basic idea in these optimizations is to batch together the processing of multiple network packets into a single 'coalesced' packet to amortize the cost of per-packet operations in the network virtualization stack that are not related to protocol processing. Receive Aggregation 'aggregates' together multiple incoming TCP packets into a single 'host' TCP packet, and Acknowledgment Offload batches together multiple TCP ACK packets into a 'large' ACK packet.

3.2.1 Receive Aggregation

The idea of Receive Aggregation is to coalesce, or 'aggregate' together multiple incoming TCP packets into a single 'host' TCP packet, and to process only the *aggregated* TCP packets in the network virtualization stack, instead of processing the individual network packets. This allows us to amortize the cost of per-packet operations in the network stack that are not related to protocol processing, such as buffer management, I/O channel forwarding, bridging, as these need to be performed only once for each aggregated TCP packet, instead of once for every network TCP packet.

Receive Aggregation is performed at the entry point of the network virtualization stack in Xen, i.e., at the point when network packets are received in the NIC driver in the driver domain. The NIC driver receives a sequence of network TCP packets from the network interface, and, instead of delivering them directly to the network stack, it first passes them to a proxy layer sitting between the driver and the network stack. This proxy layer performs the task of packet aggregation, and delivers the final aggregated TCP packets to the network bridge for further processing.

Packet aggregation is done in the proxy layer by 'chaining' together the received packets, and rewriting the TCP/IP header. The proxy layer only aggregates TCP packets that belong to the same TCP connection and which are 'in-sequence'. Aggregation is performed by maintaining a small hash table of partially aggregated TCP packets, and checking newly received TCP packets against this table to determine if they can be 'coalesced' with previously received TCP packets. When a 'sufficient' number of network TCP packets have been coalesced into an 'aggregated' TCP packet, the proxy layer delivers this packet to the bridge.

The implementation of the aggregation function is done outside of the NIC interrupt context, and is done in a work-conserving fashion. This means that no TCP packets are delayed in the aggregation hash table while the network stack is idle and ready to process packets.

Once aggregated, the TCP packet is processed in the rest of the network virtualization stack in a manner very similar to the processing of a regular TCP packet. Thus, the network bridge

in the driver domain, the backend and frontend interfaces, and the I/O channel, all process the aggregated TCP packet exactly as they would process a regular TCP packet. The only changes required are in the guest domain's network stack, in the TCP layer processing of the aggregated TCP packet (described later in this section).

Ideally, we would have liked to implement Receive Aggregation in a completely transparent manner, without any modifications to the NIC driver or the rest of the kernel, but for reasons of performance and correctness, some small changes are required to the NIC driver and in the TCP layer in the network stack of the guest domain. In the following sections, we describe in more detail different aspects of Receive Aggregation, namely, a) which packets and how many packets are chosen for Receive Aggregation, b) the structure of the aggregated TCP packet, and c) changes required to the TCP layer processing in network stack in the guest domain.

Which Packets are Aggregated

Receive Aggregation is done only for TCP packets which belong to the same TCP connection, and which are "in-sequence". Thus, the incoming packets must have the same source IP, destination IP, source port, and destination port fields. The packets must also be in sequence, both by TCP sequence number and by TCP acknowledgment number.

Packet aggregation is done only for valid TCP packets, i.e., those with a valid TCP and IP checksum. Only the IP checksum field is verified in software. For the TCP checksum we require the network interface card to support checksum verification, since it would be too expensive to do it in software. We do not aggregate TCP packets of zero length, such as pure ACK packets. Since the TCP and IP headers support a large number of option fields, it is not possible to aggregate two TCP packets if they contain different option fields. Thus, for simplicity, we only aggregate TCP packets whose IP headers do not use any IP options or IP fragmentation, and whose TCP headers use only the TCP timestamp option.

Packets which fail to match any of the conditions for Receive Aggregation are passed unmodified to the network stack. In doing so, we ensure that there is no packet reordering between

packets of the same TCP connection, i.e., any partially aggregated packet belonging to a TCP connection is delivered before any subsequent unaggregated packet is delivered.

Aggregated Packet Structure

Once a valid set of network packets is identified for aggregation, according to the conditions described above, the proxy layer coalesces them into an aggregated TCP packet for the network stack to process.

The aggregated TCP packet is created by 'chaining' together individual TCP packets to form the fragments of the aggregated packet, and by rewriting the TCP/IP header of the aggregated packet. The TCP/IP header of the first TCP fragment in the chain becomes the header of the aggregated packet, while the subsequent TCP fragments retain only their payload. The chaining is done in an OS-specific manner. In Linux, for instance, chaining is done by setting the fragment pointers in the `sk_buff` structure to point to the payload of the TCP fragments. Thus, there is no data copy involved in packet aggregation.

The TCP/IP header of the aggregated packet is rewritten to reflect the packet coalescing. The IP packet length field is set to the length of the total TCP payload (comprising all fragments) plus the length of the header. The TCP sequence number field is set to the TCP sequence number of the first TCP fragment, and the TCP acknowledgment number field is set to the acknowledgment number of the last TCP fragment. The TCP advertised window size is set to the window size advertised in the last TCP fragment. A new IP checksum is calculated for the aggregated packet using its IP header and the new TCP pseudo-header. The TCP timestamp in the aggregated packet is copied from the timestamp in the last TCP fragment of the aggregated packet.

Finally, the aggregated TCP packet is augmented with information about its constituent TCP fragments. Specifically, the TCP acknowledgment number of each TCP fragment is saved in the packet metadata structure (`sk_buff`, in the case of Linux). This information is later used by the TCP layer for correct protocol processing.

When Aggregation Stops

Network packets are aggregated as they are received from the NIC driver. The maximum number of network TCP packets that get coalesced into an aggregated TCP packet is called the Aggregation Limit. Once an aggregated packet reaches the Aggregation Limit, it is passed on to the network stack.

The actual number of network packets that get coalesced into an aggregated packet may be smaller than the Aggregation Limit, and depends on the network workload and the arrival rate of the packets. If an aggregated packet contains less than the Aggregation Limit number of network packets, and no more network packets are available for processing, then this semi-aggregated packet is passed on to the network stack without further delay. Receive Aggregation is thus work-conserving and does not add to the delay of packet processing.

We expect the performance benefits of Receive Aggregation to be proportional to the number of network packets coalesced into an aggregated packet. However, the incremental performance benefits of aggregation are expected to be marginal beyond a certain number of packets. Thus, the Aggregation Limit serves as an upper bound on the maximum number of packets to aggregate, and should be set to a reasonable value at which most of the benefits of aggregation are achieved. We determine a good cut-off value for the Aggregation Limit experimentally.

Modifications to the TCP layer

There are two modifications that are required in the guest domain's TCP layer processing, in order to handle the aggregated TCP packets correctly:

Congestion Control: Since the congestion window computed on the TCP sender side is typically updated based on the number of ACK packets received (and not on the total bytes of data acknowledged), this computation can go wrong with Receive Aggregation, where TCP packets with different ACK numbers are aggregated into a single packet. A relatively trivial modification is required to compute the congestion window correctly in the guest domain's TCP layer, by making use of the ACK numbers of all the TCP fragments of the aggregated packet.

TCP Acknowledgments: The TCP protocol specifies that an ACK packet must be generated for every alternate full TCP segment received by the receiver. If the guest domain's TCP layer considers the aggregated TCP packet as a single TCP packet, it would generate the wrong number of ACK packets. Thus, a second modification is required in the TCP layer to generate the correct number of ACK packets, based on actual the number of network TCP segments coalesced in the aggregated TCP packet.

3.2.2 Acknowledgment Offload

We now describe our second optimization for improving receive side performance in Xen guest domains.

The Acknowledgment Offload optimization allows guest domains to 'batch' together the transmission of a sequence of TCP ACK packets (belonging to the same TCP connection) into a single 'template' ACK packet. This amortizes the per-packet costs incurred in the transmission of ACK packets from the guest domain, since all operations in the network virtualization stack need to be performed only once for each template ACK packet, instead of once for every network ACK packet. Individual TCP ACK packets are generated from the template ACK packet just before the template ACK packet is queued for transmission on the physical NIC. This is done either by the NIC driver or by a proxy sitting between the network bridge and the NIC driver.

Acknowledgment Offload is an important optimization for improving receive side performance in guest domains. This is because, for a receive intensive TCP workload, at least one third of the packets processed in the network stack are ACK packets (in accordance with the TCP protocol, which mandates one TCP ACK packet to be generated for every two full TCP packets received). Thus, reducing the per-packet overheads for processing TCP ACKs provides non-trivial benefits.

Template ACK packet

The template ACK packet for a sequence of consecutive TCP ACKs is represented by the first ACK packet in the sequence, and is augmented with the ACK sequence numbers for the subsequent ACK packets (this information is stored in the `sk_buff` structure in Linux). The TCP and IP headers of the successive TCP ACK packets share most of the fields of the header, and only differ in the ACK sequence number field and the IP checksum field. Thus, the NIC driver (or a proxy) can be modified to generate the correct sequence of TCP ACK packets from the information present in the template ACK packet.

Acknowledgment Offload is preferably used in conjunction with the Receive Aggregation optimization. Receive Aggregation effectively delivers input TCP packets to the network stack in batches of aggregated packets. This allows the network stack to process a sequence of incoming TCP packets simultaneously, and make use of the Acknowledgment Offload optimization to transmit the sequence of TCP ACK packets in a batch.

3.3 Evaluation

Transmit Side Optimizations

The transmit side optimizations described in the previous sections have been implemented in Xen version 2.0.6, running Linux guest operating systems version 2.6.11.

Experimental Setup

We use a transmit micro-benchmark to evaluate the transmit networking performance of guest and driver domains. This benchmark is similar to the netperf [NET] TCP streaming benchmark, which measures the maximum TCP streaming throughput over a single TCP connection. Our benchmark is modified to use the zero-copy sendfile system call for transmit operations.

The 'server' system for running the benchmark is a Dell PowerEdge 1600 SC, 2.4 GHz Intel Xeon machine. This machine has four Intel Pro-1000 Gigabit NICs. The 'clients' for running

an experiment consist of Intel Xeon machines with a similar CPU configuration, and having one Intel Pro-1000 Gigabit NIC per machine. All the NICs have support for TSO, scatter/gather I/O and checksum offload. The clients and server machines are connected over a Gigabit switch.

The experiments measure the maximum transmit throughput achievable with the benchmark running on the server machine. The server is connected to each client machine over a different network interface, and uses one TCP connection per client. We use as many clients as required to saturate the server CPU, and measure the throughput under different Xen and Linux configurations. All the profiling results presented in this section are obtained using the Xenoprof system-wide profiler in Xen [MST+05].

Overall Results

We evaluate the following configurations: 'Linux' refers to the baseline unmodified Linux version 2.6.11 running native mode. 'Xen-driver' refers to the unoptimized XenoLinux driver domain. 'Xen-driver-opt' refers to the Xen driver domain with our optimizations. 'Xen-guest' refers to the unoptimized, existing Xen guest domain. 'Xen-guest-opt' refers to the optimized version of the guest domain.

Figure 3.2 compares the transmit throughput achieved under the above 5 configurations.

For the transmit benchmark, the performance of the Linux, Xen-driver and Xen-driver-opt configurations is limited by the network interface bandwidth, and does not fully saturate the CPU. All three configurations achieve an aggregate link speed throughput of 3760 Mb/s. The CPU utilization values for saturating the network in the three configurations are, respectively, 40%, 46% and 43%.

The optimized guest domain configuration, Xen-guest-opt improves on the performance of the unoptimized Xen guest by a factor of 4.4, increasing transmit throughput from 750 Mb/s to 3310 Mb/s. However, Xen-guest-opt gets CPU saturated at this throughput, whereas the Linux configuration has a CPU utilization of only 40% to saturate 4 NICs.

We now examine the contribution of individual optimizations for the transmit workload. Fig-

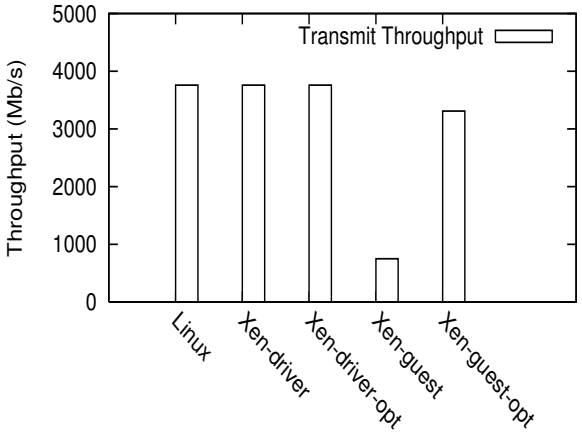

Figure 3.2: TCP transmit throughput under different configurations. The optimized Linux guest domain achieves throughput very close to native Linux throughput.

ure 3.3 shows the transmit performance under different combinations of optimizations. 'Guest-none' is the guest domain configuration with no optimizations. 'Guest-ioc' is the guest domain configuration using only the I/O channel optimization. 'Guest-high' uses only the high level virtual interface optimization. 'Guest-high-ioc' uses both high level interfaces and the optimized I/O channel.

The greatest contribution to guest transmit performance comes from the use of a high-level virtual interface (Guest-high configuration). This optimization improves guest performance by 272%, from 750 Mb/s to 2794 Mb/s. The I/O channel optimization yields an incremental improvement of 439 Mb/s (15.7%) over the Guest-high configuration to yield 3230 Mb/s (configuration Guest-high-ioc).

Benefits of a High level Interface

We explain the improved performance resulting from the use of a high level interface in figure 3.4. The figure compares the execution overhead incurred for running the transmit workload on a single NIC (in millions of CPU cycles per second) in the guest domain, driver domain and the Xen VMM, compared between the high-level Guest-high configuration and the Guest-none

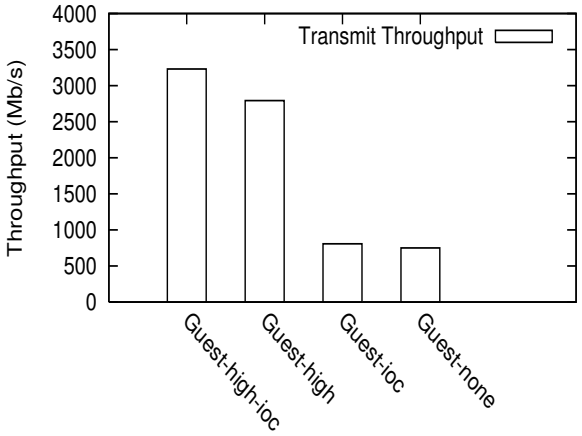

Figure 3.3: Contribution of individual transmit side optimizations to a Linux guest domain's transmit performance. The biggest improvement comes from the use of a high level virtual interface.

configuration.

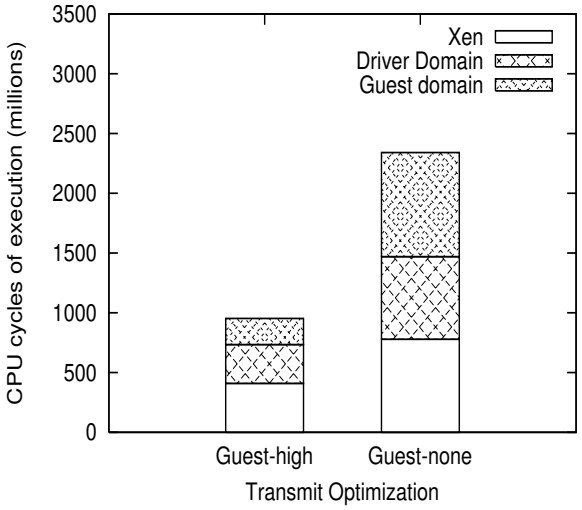

Figure 3.4: Breakdown of the execution costs for TCP transmit workload with and without the high level virtual interface optimization. With a high-level interface, there is a reduction in overhead on all components of the virtualization path.

The use of the high level virtual interface reduces the execution cost of the guest domain by almost a factor of 4 compared to the execution cost with a low-level interface. Further, the

high-level interface also reduces the Xen VMM execution overhead by a factor of 1.9, and the driver domain overhead by a factor of 2.1. These reductions are largely due to the reduced per-packet overheads in the network virtualization stack, and the support for scatter/gather DMA in the virtual NIC, which allows TCP transmits to be truly zero-copy.

The use of a high level virtual interface gives performance improvements for the guest domain even when the offload features are not supported in the physical NIC. Figure 3.5 shows the performance of the guest domain using a high level interface with the physical network interface supporting varying capabilities. The capabilities of the physical NIC form a spectrum, at one end the NIC supports TSO, SG I/O and checksum offload, at the other end it supports none of the offload features, with intermediate configurations supporting some offload capabilities.

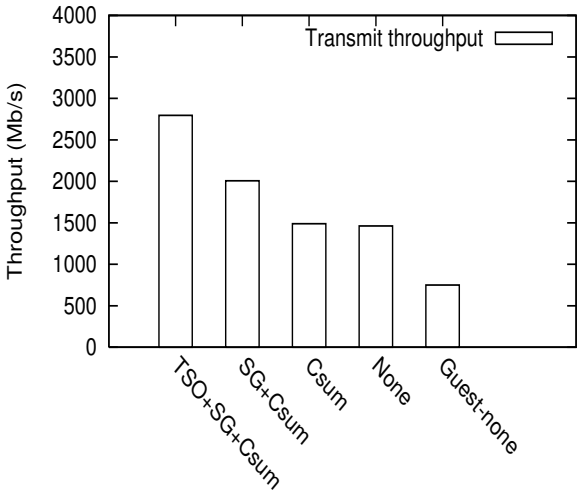

Figure 3.5: Advantages of the HLVI interface for TCP transmit workloads in Xen guest domains. Even when the physical NIC does not support offload, HLVI yields performance benefits.

Even in the case when the physical NIC supports no offload feature (bar labeled 'None'), the guest domain with a high level interface performs nearly twice as well as the guest using the default interface (bar labeled Guest-none), viz. 1461 Mb/s vs. 750 Mb/s. Thus, even in the absence of offload features in the NIC, the reductions in per-packet overheads achieved through the use of a high level interface in the guest domain leads to improved transmit performance.

Non Zero-copy Transmits

So far we have shown the performance of the guest domain when it uses a zero-copy transmit workload. This workload benefits from the scatter/gather I/O capability in the network interface, which accounts for a significant part of the improvement in performance when using a high level interface. We now show the performance benefits of using a high level interface, and the other optimizations, when using a benchmark that uses copying writes instead of the zero-copy sendfile. Figure 3.6 shows the transmit performance of the guest domain for this benchmark under the different combinations of optimizations.

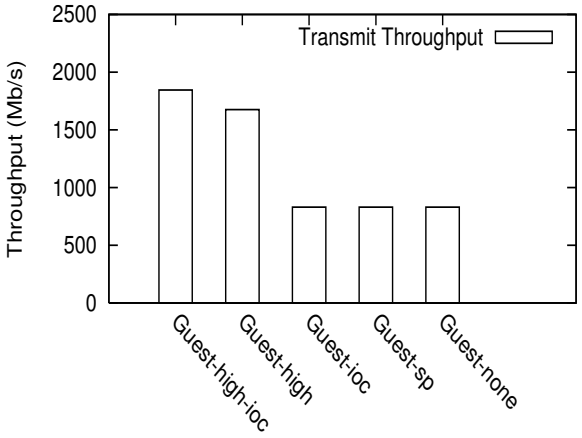

Figure 3.6: Transmit performance of Xen guest domains using an HLVI interface for non zero-copy workloads

The breakup of the contributions of individual optimizations in this benchmark is similar to that for the sendfile benchmark. The best case transmit performance in this case is 1848 Mb/s, which is much less than the best sendfile throughput (3310 Mb/s), but still significantly better than the unoptimized guest throughput.

Receive Side Optimizations

We have implemented Receive Aggregation and Acknowledgment Offload in a stock 2.6.16.34 Linux kernel, and in the Xen VMM version 3.0.4 running Linux 2.6.16.38 guest operating systems.

In this evaluation, we show that Receive Aggregation and Acknowledgment Offload are effective at improving receive side performance even in a native Linux environment. Although we have performed extensive analysis of the receive side processing overheads in a native Linux system, in this section, we restrict ourselves to the performance analysis in a virtualized environment. More details of our work can be found in [MZ08].

We first evaluate the performance benefits of the receive optimizations for three systems: a Xen guest operating system, a uniprocessor Linux system, and an SMP Linux system. Next, we experimentally determine a good cut-off value for the Aggregation Limit. Finally, we demonstrate that our optimizations do not affect the performance of latency-sensitive workloads.

Performance Benefits

We use a receive microbenchmark to evaluate the TCP receive performance of the system under test. This microbenchmark is similar to the netperf [NET] TCP streaming benchmark and measures the maximum TCP receive throughput which can be achieved over a single TCP connection.

The server machine used for our experiments is a 3.0 GHz Intel Xeon machine, with 800 MHz FSB and 512 MB of DDR2-400 memory. The machine is equipped with five Intel Pro1000 Gigabit Ethernet cards, fitted in 133 MHz, 64 bit PCI-X slots. We run one instance of the microbenchmark for each network card. The 'receiver' end of each microbenchmark instance is run on the server machine and the 'sender' end is run on another client machine, which is connected to the server machine through one of the Gigabit NICs. The sender continuously sends data to the receiver at the maximum possible rate, for the duration of the experiment (60s). The final throughput metric reported is the sum of the receive throughput achieved by

all receiver instances.

Overall Results

Figure 3.7 shows the overall performance benefit of using Receive Aggregation and Acknowledgment Offload in the three systems. The figure compares the receive performance of the three systems (throughput, in Mb/s) with and without the use of the receive optimizations. The 'Linux UP' histograms show the performance for the uniprocessor Linux system, 'Linux SMP', for the SMP Linux system, and 'Xen', for the Linux guest operating system running on the Xen VMM.

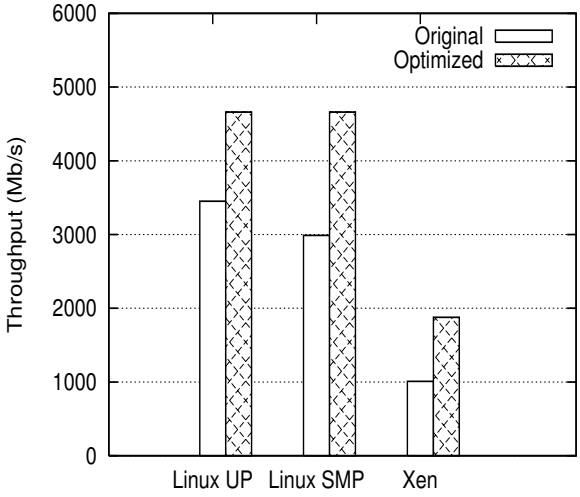

Figure 3.7: Overall performance improvements for a TCP receive workload in three configurations: a Linux uniprocessor system, a Linux SMP system and a Linux guest domain running on the Xen VMM.

The performance results for the three systems are as follows. For the uniprocessor Linux system, the unmodified (Original) Linux TCP stack reaches full CPU saturation at a throughput of 3452 Mb/s. With the use of the receive optimizations, the system (Optimized) is able to saturate all the five Gigabit network links, to reach a throughput of 4660 Mb/s. The CPU is still not fully saturated at this point and is at 93% utilization. The system is thus constrained by the number of NICs, and with more NICs, it can theoretically reach a (CPU-scaled) throughput

of 5050 Mb/s. The performance gain of the system is thus 35% in absolute units and 45% in CPU-scaled units.

For the SMP Linux system, the performance of the Original system is 2988 Mb/s, whereas the optimized system is able to saturate all five NICs to reach a throughput of 4660 Mb/s. As in the uniprocessor case, the optimized SMP system is still not CPU saturated and is at 93% CPU utilization. Thus, the performance gain in the SMP system is 55% in absolute terms and 67% in CPU-scaled units.

For the Linux guest operating system running on Xen, the unoptimized (Original) system reaches full CPU saturation with a throughput of only 1088 Mb/s. With the receive optimizations, the throughput is improved to 1877 Mb/s, which is 86% higher than the baseline performance. In both cases, the CPU is at full saturation.

The contribution of Acknowledgment Offload to the above performance improvements is non-trivial. Using just Receive Aggregation without Acknowledgment Offload, the performance improvement to the three configurations is, respectively, 26%, 36% and 45%, with CPU utilization reaching 100% in all three cases.

Analysis of the Results

We can better understand the performance benefits of the receive optimizations by comparing the overhead profiles of the network virtualization stack, with and without the use of the optimizations. Figure 3.8 shows the breakdown of the receive processing overhead in the Xen guest domain, with and without receive optimizations, in terms of CPU cycles incurred per packet. The different profile categories shown here are identical to the ones discussed in chapter 2, except that we have one extra category `aggr`, which measures the overhead of doing Receive Aggregation.

Receive Aggregation and Acknowledgment Offload effectively reduce the number of packets processed in the network stack by a factor of up to 20, which is the Aggregation Limit in our system. This greatly reduces the overhead of all the per-packet components in the network

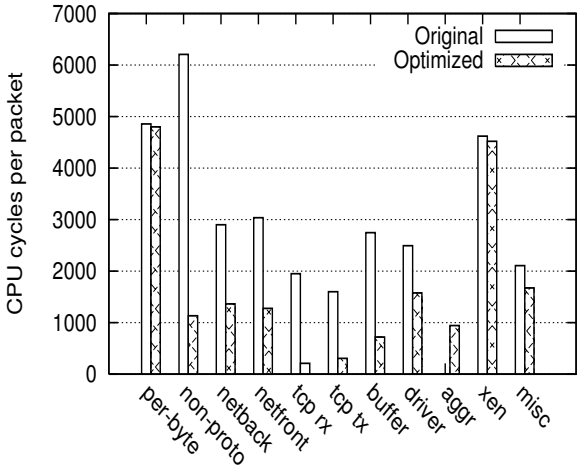

Figure 3.8: Breakdown of receive processing overheads for a Linux guest domain running on the Xen VMM. Receive aggregation and Acknowledgment offload greatly reduce the overhead of per-packet operations.

stack.

Thus, the total overhead of the per-packet routines (`non-proto`, `netback`, `netfront`, `tcp rx`, `tcp tx` and `buffer`) in the network virtualization stack is reduced by a factor of 3.7 with the use of the receive optimizations. The greatest visible reduction is in the overhead of the `non-proto` routines, which includes the bridging and netfilter routines in the driver domain and the guest domain. The overheads of the `netfront` and `netback` paravirtual drivers are reduced to a lesser extent, primarily because they incur a per-TCP fragment overhead instead of a purely per-packet overhead. Other per-packet components, such as the TCP receive and transmit routines (`TCP rx` and `TCP rx`) and buffer management (`buffer`) also show significant reduction in overhead.

The overhead of Receive Aggregation (`aggr`) itself is small compared to the other overheads.

Choosing the Aggregation Limit

The performance benefit of Receive Aggregation is proportional to the number of TCP packets which are combined to create the aggregated host TCP packet. A greater degree of aggregation

results in a greater reduction in the per-packet overhead. However, beyond a limit, packet aggregation does not yield further benefits. We determine a good cut-off value for this Aggregation Limit experimentally.

Figure 3.9 shows the total CPU execution overhead (in CPU cycles per packet) incurred for receive processing in a Linux system, as a function of Aggregation Limit. (We expect qualitatively similar graphs for other systems, including the Xen guest domain.)

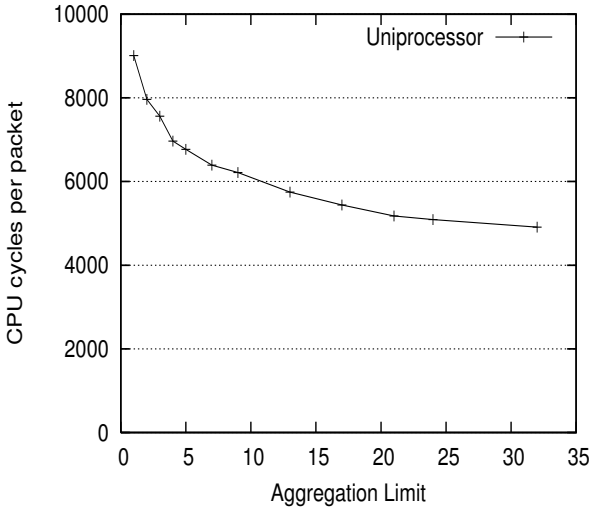

Figure 3.9: Experimentally determining the Aggregation Limit to use with Receive Aggregation. Even a small aggregation limit significantly reduces network processing overhead.

Increasing the Aggregation Limit initially yields a sharp reduction in the CPU processing overhead of packets. The figure shows that most of the benefits of Receive Aggregation can be achieved with a relatively small Aggregation Limit. We choose a value of 20 for the Aggregation Limit as it can be seen that additional aggregation does not yield any substantial improvement.

The Aggregation Limit measured above can also be derived analytically, since it only depends on the percentage overhead of the per-packet operations whose overhead can be scaled down by aggregation. For instance, if `x`% of the overhead is constant, and `y`% is the per-packet overhead that can be reduced by aggregation (with `x + y = 100`), then using an aggregation factor of `k` should reduce the system CPU utilization from `x + y` to `x + y/k`. Figure 3.9 appears to match the plot of `x + y/k` as a function of `k` fairly well. This gives us confidence that the

	Requests/sec (Original)	(Optimized)
Linux UP	7874	7894
Linux SMP	7970	7985
Xen	6965	6953

Table 3.1: Impact of Receive aggregation and Acknowledgment offload on latency critical applications

Aggregation Limit chosen is quite robust and not arbitrary, and will hold across a number of different systems.

Impact on Latency Sensitive Workloads

We use the netperf [NET] TCP Request/Response benchmark to evaluate the impact of the receive optimizations on the latency of packet processing.

This benchmark measures the interactive request-response performance of a client and server program connected by a TCP connection. The client sends the server a one-byte 'request', and waits for a one-byte 'response' from the server. On receiving the response, the client immediately sends another request. The benchmark measures the maximum request-response rate achieved between the client and the server.

Table 3.1 compares the performance of the three systems on the TCP Request/Response benchmark. The table shows that our receive optimizations have no noticeable impact on the latency of packet processing in the network stack.

This is because of the work-conserving nature of Receive Aggregation. Since there is only one network packet to process at a time, no packet aggregation is done, and the packet is passed on to the network stack immediately to prevent it from being idle.

3.4 Summary and Discussion

In this chapter, we described transmit and receive optimizations the can be applied to the existing Xen I/O architecture. The basic idea of these optimizations was to focus on reducing the per-packet overhead of packet processing in the network stack, by using 'batching' or 'coalescing'

techniques that reduce the number of packets that the network stack has to process. These optimizations are successful at improving overall network performance by amortizing the cost of per-packet operations in the network virtualization stack that are not relevant to protocol processing, such as buffer management, I/O channel transfers and bridging operations.

The optimizations presented in this chapter are best suited for transmit and receive workloads which are *traffic-intensive*, i.e., workloads which have a large amount of traffic flowing on a small number of TCP connections. For other workloads, such as workloads which process a large number of short-lived, low traffic connections, these optimizations are likely to be less effective. This is because in our optimizations, we perform packet coalescing only when the packets belong to the same TCP connection. Thus, our optimizations are able to reduce per-packet overhead only when the workload consists of traffic intensive TCP connections.

Another 'limitation' of our optimizations is that they do not fundamentally alter the network virtualization architecure used, i.e., we still use the Xen driver domain architecture. As discussed in chapter 2, an important limitation of this architecture is that frequent context switches are incurred between the guest and the driver domain for network I/O operations, which leads to performance degradation. In the following two chapters, we present solutions which eliminate the driver domain from the performance-critical path of network operations, either by running the drivers directly in the hypervisor (chapter 4), or by performing the task of virtualization in the network interface itself (chapter 5).

Chapter 4

TwinDrivers

In this chapter, we present the TwinDrivers framework, which allows us to run device drivers directly in the hypervisor in a *safe* and *efficient* manner.

The Xen VMM uses the driver domain architecture for reasons of safety and to reduce the software engineering effort required to develop and maintain device drivers. By running the drivers in the driver domain, a bug in the driver does not compromise the hypervisor or other VMs. Furthermore, it avoids having to (re)implement either the device driver, or the driver support infrastructure in the hypervisor. Instead, the driver simply re-uses the driver support infrastructure already present in the driver domain.

However, as discussed in chapters 2 and 3, the drawback of this approach is loss of performance. Since the device driver is located in a different driver VM, and thus a different address space than the guest VM, extra context switching overhead is incurred in invoking the device driver and in interrupt handling.

The alternative is to use a hypervisor based virtualization architecture, in which the device drivers run directly in the hypervisor. This approach gives better performance because it avoids context switches for calls between the hypervisor driver and the guest VM. Unfortunately, this approach requires the entire driver and its support library to be either developed anew for the hypervisor, or to be ported from an existing operating system. Both approaches incur a

significant software development effort. In addition, this approach also leaves the hypervisor vulnerable to bugs in the device driver.

The TwinDrivers approach tackles the tradeoff between performance on the one hand and safety and reduction in coding effort on the other hand. Our goal is to combine the performance benefits of the hypervisor-based driver approach with the safety and software engineering benefits of the VM-based driver approach.

We take a driver developed for a guest operating system, such as Linux, and we *semi-automatically* produce from it, by binary rewriting, a driver that *efficiently and safely* runs in the hypervisor. At runtime, two instances of the driver are run at the same time: The original one, which we call the VM instance, runs in a VM. The derived one, which we call the hypervisor instance, runs in the hypervisor. The hypervisor instance takes care of performance-critical operations of the device driver. For instance, for a network card driver, this includes transmitting and receiving packets. The VM instance takes care of the other operations such as device configuration, management, error handling, etc.

Although there are two separate instances of the driver running, there is only a single instance of the driver data, residing in the VM address space. The hypervisor instance accesses only the driver and VM data structures in the VM, and does not access any hypervisor data structures. From this simple rule derives the *safety* of the approach: The hypervisor instance cannot access, and therefore cannot corrupt the hypervisor data structures.

Although the driver data is located in the VM address space, the hypervisor instance can access this data while running in any guest VM context by using an address translation mechanism called *Software Virtual Memory* (SVM). This allows the hypervisor to invoke its driver instance while running in any guest context without switching address spaces. This is the key to achieving good *performance*.

Keeping only a single copy of the data in the VM address space also allows the hypervisor instance to invoke the driver support routines in the VM for operations on these data structures. This is done through an *upcall* mechanism from the hypervisor to the VM. The *upcall* approach

avoids the implementation in the hypervisor of the entire set of driver support routines. Instead, the hypervisor only implements a small set of performance-critical support routines needed to achieve good performance. This is the key to reducing the *software engineering effort* to support the hypervisor driver instance.

Although, we have implemented the ideas described above in the the Xen hypervisor for Linux network drivers, but we believe the ideas are generally applicable. We have used our binary rewriting system to twin the Intel e1000 driver. The TwinDrivers system allows us to improve the Xen guest domain networking throughput in CPU-scaled units by a factor of 2.4 for transmit workloads, and by a factor of 2.1 for receive workloads. The resulting throughput is also within 64 to 67% of native Linux throughput.

The outline of the rest of this chapter is as follows. Section 4.1 presents the principles underlying the TwinDrivers approach. Section 4.2 presents in more detail the design of the TwinDrivers approach, and Section 4.3 presents our current implementation. Section 4.4 presents our performance results for network I/O.

4.1 Principles

We discuss the design and principles underlying the TwinDrivers approach in the context of the Xen VMM, using the example of a network interface card driver. Figure 4.1 shows the overall architecture of the TwinDrivers approach. TwinDrivers uses two driver instances, one running in dom0 and one running in the hypervisor, but only one instance of the driver data, residing in dom0.

4.1.1 Two Driver Instances

We take a device driver from the Linux driver domain, and rewrite the binary to produce a driver that can execute in the hypervisor. At runtime, two instances of the driver are run at the same time: the original VM driver instance runs in dom0, and the derived hypervisor driver runs in the hypervisor. We first load the VM driver into the dom0 kernel where it performs the

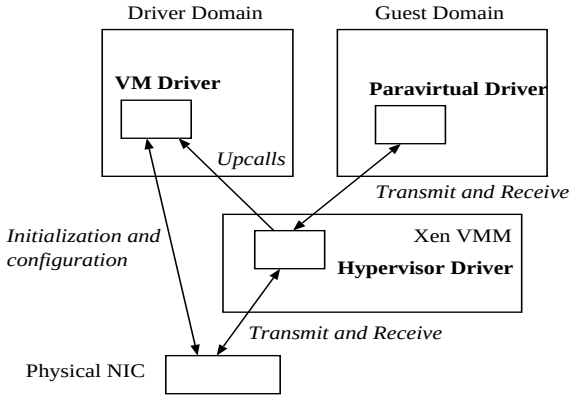

Figure 4.1: TwinDrivers Architecture

initialization of the NIC and the driver data structures. After the initialization is complete, we load the hypervisor driver into the Xen hypervisor. TwinDrivers uses this driver for performing the performance-critical send and receive operations on the NIC.

We develop a new paravirtual driver for guest domains, which interfaces with the Xen hypervisor through a hypercall interface and allows it to invoke the hypervisor driver to transmit and receive packets on its behalf. No context switch is incurred in invoking the hypervisor driver from the guest context.

The VM driver instance continues to run in dom0 to provide support for all other NIC operations which are not performance critical. These include reconfiguring NIC parameters using `ethtool`-like tools, doing periodic error checks on the NIC using timers, collecting and reporting device statistics, etc. Keeping the VM driver instance running in the driver domain for these functions allows us to restrict the hypervisor interface to the driver to just the transmit and receive functions, and avoids the need to port existing user-space tools (such as `ethtool`) to use the new hypervisor driver instead of the VM driver.

4.1.2 Single Instance of Driver Data Structures

In the TwinDrivers architecture, although there are two instances of the driver running, there is only a single instance of the driver data structures residing in the dom0 address space. The

hypervisor driver instance accesses the shared dom0 driver data structures for all its operations.

We introduce a new mechanism called *Software Virtual Memory* (SVM) that allows the hypervisor instance to access the dom0 data structures from any guest domain address space. Software Virtual Memory is a runtime address translation and protection mechanism which is incorporated into the hypervisor driver during binary rewriting of the VM driver.

The SVM mechanism is the key to combining *efficiency* and *safety* in the hypervisor driver. By allowing the hypervisor instance to access the dom0 driver data structures from any guest domain context, expensive context switches are avoided on driver invocation, and high *performance* is achieved. By restricting all memory accesses from the hypervisor instance to the dom0 address space, *safety* is achieved and the hypervisor is protected from memory corruption bugs in the driver.

A third advantage of keeping a single copy of all data in dom0 address space is that it allows the hypervisor instance to reuse driver support routines present in the dom0 kernel. The driver support routines form a large body of code in the VM (Linux) kernel, and it requires significant engineering effort to provide identical support routines in the Xen hypervisor in order to run the dom0 drivers [LUSG04].

However, since the driver data resides in the dom0 address space, the hypervisor instance can reuse the driver support routines in the VM using an *upcall* mechanism. The *upcall* mechanism allows the hypervisor to avoid having to implement the majority of the driver support routines which are invoked only infrequently by the driver. Instead, the hypervisor implements only a small set of performance-critical support routines which are needed for good performance. The *upcall* mechanism thus allows us to reduce the *software engineering effort* needed to support the driver while retaining the performance benefits.

We now describe these mechanisms in more detail in the following sections.

4.2 Detailed Design

4.2.1 Software Virtual Memory

Software Virtual Memory (SVM) is the key mechanism that enables the hypervisor driver instance to access the driver data residing in dom0 address space. SVM uses a combination of runtime virtual address translation and page remapping to allow memory accesses to the dom0 address space from the hypervisor without a context switch, and to prevent invalid access to the hypervisor address space.

At the core of the SVM mechanism is a *Software translation table* (`stlb`) which maps from virtual memory page addresses in dom0 address space to *mapped* virtual page addresses in the hypervisor address space. The *mapped* page address in an `stlb` entry is a hypervisor page which maps to the same physical page as the corresponding dom0 page address.

To produce the hypervisor driver, every instruction which references memory locations in the original VM driver is rewritten to make use of SVM to perform the memory access (except for stack-relative memory references). Thus, at runtime, every memory access to dom0 address space from the hypervisor driver instance is first translated using the `stlb` table into a *mapped* address, and the memory access is made using the translated address. Attempts to access the hypervisor address space by the driver are detected and prevented because the `stlb` table does not contain valid mappings for hypervisor addresses. On such an illegal memory access by the driver, it is aborted.

Figures 4.2 and 4.3 give an example of how rewriting works, using an example (figure 4.2) of an indirect memory reference instruction which loads the value at the memory location in `r_src` into the register `r_dest`. Figure 4.3 shows how the rewritten code translates the address using the `stlb` table and uses the translated address to load the value. [1]

The `stlb` table acts as a hashtable storing translations from dom0 virtual page addresses to the *mapped* virtual page addresses in the hypervisor. In lines 1 to 6, the lower 12 bits of the

[1] We discuss how the instruction rewriting works for more complicated x86 instructions in section 4.3.

```
        movl    %(r_src), %r_dest
```

Figure 4.2: Indirect memory reference in original code

```
1.      leal    %(r_src),       %r1
2.      movl    %r1,            %r2
3.      andl    0xfffff000,     %r1
4.      movl    %r1,            %r3
5.      andl    0xfff000,       %r1
6.      shrl    $9,             %r1
7.      cmpl    stlb(%r1),      %r3
8.      jne     .L_slow_path
9.      xorl    4+stlb(%r1),    %r2
10.     movl    (%r2),          %r_dest
```

Figure 4.3: Rewritten code using SVM

dom0 page address (for a 32 bit system) are used as an index into the `stlb` table. In line 7, we check if the indexed `stlb` entry for the dom0 page is valid, i.e., if the page has been previously mapped into the hypervisor address space (and there are no hash collisions). If so, the `stlb` entry is used to compute the final translated address and this address is used for the memory reference (lines 9 and 10).

If the `stlb` translation entry for the virtual address accessed is not valid (line 8), control is transferred to a slow-path lookup routine. If the `stlb` lookup failed because of a hash collision, the slow-path routine looks up a hash chain and fills in the correct mapped virtual page address.

If, however, the lookup failed because the virtual address was being accessed for the first time, the slow-path routine checks the permissions of the memory access and, if the access is permitted (i.e., the memory page belongs to dom0 address space), it creates a new hypervisor mapping for the dom0 address. It allocates a new hypervisor virtual page, and maps it to the physical page corresponding to the accessed page. It then fills in the `stlb` table with the new translation entry. Subsequent accesses to this dom0 page are translated directly from the `stlb` table. [2] Entries in the `stlb` table are thus dynamically filled in as memory accesses to dom0

[2] Actually, two consecutive dom0 pages are mapped into the hypervisor for each `stlb` 'miss'. This is because the Intel instruction set permits unaligned memory accesses, so a memory access may straddle two pages.

address space are made by the hypervisor driver instance. In our implementation, we use an `stlb` hashtable with 4096 entries, mapping up to 16MB of dom0 virtual memory.

The fast path of the SVM-based memory access replaces one memory instruction in the original code with ten instructions in the rewritten code. [3] While this may seem prohibitive at first, in practice its impact on overall performance is much smaller. Firstly, in a typical driver, only roughly 25% of the instructions reference memory, and are rewritten to use SVM (we measured this for some network drivers). Secondly, in a typical network-intensive workload, the device driver itself incurs roughly 10-15% of the total overhead. As we show in section 4.4, the overall performance impact of using an SVM-based device driver is quite small.

The `stlb` based SVM memory access is not used for stack-relative memory accesses (i.e., it is used only to translate heap memory access and not stack memory access). This is because the hypervisor driver instance uses a separate stack of its own in the hypervisor address space, and overflow on this stack is prevented by the use of guard pages.

4.2.2 Upcalls from the hypervisor into dom0

The hypervisor uses the upcall mechanism to reuse the driver support routines present in dom0. An upcall is a synchronous, cross-address-space function invocation and return mechanism. Upcalls are used by the hypervisor to link infrequently called support routines from the driver to the corresponding routines in the driver VM using special stub routines in the hypervisor. On a call to a stub routine by the hypervisor driver, the stub routine first saves the parameters of the call and then initiates an *upcall* into the driver domain by sending a special synchronous virtual interrupt to dom0. If the support routine is invoked while the driver is running in a guest domain context, a synchronous context switch to dom0 is done first. Additionally, before the virtual interrupt is sent to dom0, the stub routine also switches from the hypervisor stack to an 'upcall' stack. This is because, in the Xen hypervisor, the state of the hypervisor stack is

[3] Additional scratch registers are needed for computing the address translation, which may require spilling some registers to memory, and can increase the length of the fast path. However, we avoid the cost of spilling registers most of the time by doing a register liveness analysis to determine the set of free registers available at each instruction.

not saved on transition to the guest domain.

An upcall handler is registered in the driver domain to receive upcall requests via synchronous virtual interrupts. It recovers the upcall parameters, sets up the stack and register parameters, and then invokes the driver support routine. On return from the driver support routine, the upcall handler saves the return values of the routine and 'returns' to the stub routine via a hypercall. The stub routine eventually returns to the hypervisor driver with the support routine's return values (possibly after doing another domain switch back to the guest domain).

For the upcall mechanism to work correctly, the environment in which the driver support routine is called from the upcall handler in dom0 must be identical to the environment in which it is called from the hypervisor driver. The call environment of the routine comprises of three components: the heap, the stack and the registers. The heap environment is identical in the two invocations because there is a single driver data instance which resides in dom0 address space. The register values for the two calls are made identical by the upcall mechanism. Although the stack parameters passed are identical in the two cases, the stack address is different. This could be problematic, for instance, if the hypervisor driver passes addresses of its stack variables as parameters to the dom0 support routines. In this case, the dom0 support routine would try to dereference a hypervisor driver stack address and would cause a protection fault. One possible solution would be to use instruction emulation to trap and emulate the access from the dom0 support routine to the hypervisor driver stack, after making appropriate validity checks. In practice, since it is uncommon to pass stack variables by reference, we have not encountered the stack dereference problem for any upcall to driver support routines from network drivers. Thus, currently we have not implemented the proposed solution.

4.2.3 Support routines in the hypervisor

Upcalls can be expensive because they potentially involve a context switch and transition to the driver domain. To avoid the cost of an upcall on invocation of every driver support routine

called by the driver, the hypervisor provides implementations of some support routines which are frequently called during the execution of performance-critical parts of the driver. For a typical driver, the set of such routines is a small fraction of the total number of support routines that are called by the driver.

For instance, table 4.1 lists the Linux driver support routines that are called during error-free execution of the transmit and receive routines of the Intel e1000 driver. There are only 10 such functions, compared to the 97 routines called by the e1000 driver for all its operations.

Routine name	*Description*
__netdev_alloc_skb	*allocate sk_buffs*
dev_kfree_skb_any	*free sk_buffs*
netif_rx	*receive network packets*
dma_map_single	*map DMA buffer*
dma_map_page	*map DMA page*
dma_unmap_single	*unmap DMA buffer*
dma_unmap_page	*unmap DMA page*
_spin_trylock	*acquire spinlock*
_spin_unlock_irqrestore	*release spinlock, restore interrupts*
eth_type_trans	*process MAC header*

Table 4.1: Functions called frequently from the e1000 network driver

The support routines which are implemented in the hypervisor make use of the `stlb` translation table explicitly while accessing driver data in dom0 address space. For support routines that need to allocate and free memory in the dom0 heap, such as `__netdev_alloc_skb` and `dev_kfree_skb_any`, we use a preallocated pool of buffers from dom0 heap which are reserved for use by the hypervisor routines. We use a simple reference counter trick to prevent other routines in the dom0 kernel from accessing these buffers.

4.2.4 Synchronization

Concurrent access to the shared data instance from the hypervisor and VM driver instances introduces the issue of synchronization. Fortunately, this is easily resolved. If the original driver is compiled for an SMP environment, then it already uses the correct synchronization

primitives to access shared data. These synchronization operations continue to work correctly for the hypervisor driver instance since they operate on atomic synchronization variables which are also shared between the hypervisor and VM driver.

Disabling interrupts is a common synchronization mechanism used when sharing data structures between the device driver and the operating system. Since the original VM driver runs inside dom0, the dom0 kernel masks and unmasks a *virtual interrupt* flag instead of the real CPU interrupt flag, when it wants to prevent the driver interrupt handler from running. Thus, the hypervisor must respect the *virtual interrupt* flag of the dom0 kernel before invoking the interrupt handler of the hypervisor driver. This is ensured by invoking the hypervisor driver interrupt handler routine in a schedulable 'softirq' context, instead of directly in the interrupt context.

4.2.5 Safety of Derived Hypervisor Driver

The SVM mechanism ensures memory safety of the derived hypervisor driver. Since every heap access from the hypervisor driver is translated before the access is made, invalid accesses to the hypervisor address space, or to other domain memory, are detected and prevented by SVM.

Although the derived hypervisor driver is secure against the most common kind of driver bugs, namely memory corruption bugs, it still suffers from some safety issues that are already present in the current Xen driver domain architecture. Specifically, since the network driver has full, privileged access to the network interface, a buggy or malicious driver can set up illegal DMA transfers that allow it to read from or write to memory regions it is not allowed to access. This is a safety violation that already exists with the current Xen driver domain model, where the dom0 driver has privileged access to the NIC. A complete solution to this problem requires the use of an IOMMU that can be programmed to restrict the memory regions accessible from the network card.

There are some additional unsafe situations that are not currently handled in the Twin-Drivers framework. However, these can be handled using existing mechanisms. We describe

some of these below.

Stack Corruption

Currently, the SVM memory protection mechanism is applied only to heap accesses, and not for stack-relative accesses. This is done because the hypervisor driver does not require address translation in order to access its hypervisor stack. However, this mechanism is not sufficient to prevent stack corruption errors. For instance, a buffer overflow error in the hypervisor driver can cause the driver to return to an invalid address, which is a security violation.

The stack corruption problem can be addressed by using SVM-like mechanisms to insert checks in the hypervisor driver to ensure the safety of stack-relative memory accesses. These checks are required only for those memory accesses that cannot be statically determined to be safe. For instance, accesses to constant offsets from the stack pointer can be potentially statically verified. For the small number of variable-offset accesses from the stack pointer, additional validity checks would need to be inserted. Alternatively, the problem of control flow integrity can also be solved using techniques similar to those used in XFI [EAV+06].

Non-memory related errors

The TwinDrivers framework does not currently handle non-memory related errors in the hypervisor driver. For instance, if the hypervisor driver goes into a deadlock or an infinite loop, it can prevent the hypervisor from regaining control. Such resource hoarding bugs can be prevented by mechanisms similar to those used, for instance, in VINO [SESS96]. The VINO extensible kernel makes use of timeouts to limit the duration of execution of the extension code. Similar mechanisms can be used to limit the execution time of the hypervisor driver.

Another category of bugs that is not currently handled is the use of privileged instructions, such as modifying the page tables to corrupt the system. These kinds of bugs can be detected and prevented by static inspection of the driver code during binary translation.

4.3 Implementation

4.3.1 Deriving the hypervisor driver

The hypervisor driver is created by binary rewriting of the VM driver. In the first step, we produce the assembly file of the VM driver, either by disassembling the VM driver binary, or, if the driver source is available, by directly compiling the driver into assembly. Since we work with Linux drivers, which are available in source form, we take the latter approach.

This VM driver assembler file is fed into an assembler-level rewriting tool, which generates the hypervisor assembler file as output. Conceptually, assembler-level rewriting is equivalent to binary rewriting, although working at the assembly level significantly simplifies the implementation of parsing and code generation. The hypervisor assembler file generated is eventually compiled into the hypervisor driver binary.

The rewriting tool performs a set of transformations to incorporate the SVM mechanism into the hypervisor driver. For memory reference instructions in the VM driver, the transformation applied is described in section 4.2.1. We now describe the transformations for other x86 instructions in the VM driver that reference memory in more complex ways.

String instructions

The x86 instruction set contains a number of 'string' instructions that can be used to perform string operations on blocks of contiguous data in memory, such as copying, string comparison, etc. Examples of such instructions include `movs`, `cmps`, `lods`, `stos`, `scas`, etc. These instructions take as operands the source and/or destination memory address, and an implicit length operand. For instance, the `rep; movs` instruction copies `ecx` bytes of data from source address `esi` to destination address `edi`.

When translating such instructions to use the SVM mechanism, it is not sufficient to simply translate the source and destination address operands. This is because the string operands of these instructions may span multiple pages, whereas the `stlb` translations for the string ad-

dresses may not necessarily map the contiguous dom0 pages containing the string to contiguous hypervisor pages.

Thus, for translating string instructions, we generate code that loops over the entire string in chunks of page length, and use the string instruction on the individual string chunks that are guaranteed to lie within a single page. Within the loop body, the regular address translation mechanism is used for the starting string source and destination addresses.

Indirect calls

The x86 instruction set allows routines to be 'indirectly' called by specifying the address of the routine as a register or memory operand. For example, the instruction '`call %eax`' makes an indirect call to the routine whose address is given in the `eax` register. Since all data is shared between the hypervisor driver and the VM driver (including the values of function pointers), the address of the indirectly called routine in the hypervisor driver actually points to the routine in the VM driver.

Thus, for indirect calls, the address of the called VM driver routine is first translated to the address of the corresponding hypervisor driver routine, and then the actual call is made. Similar to the `stlb` table for memory addresses, an `stlb_call` table caches translations from VM-driver routine addresses to hypervisor-driver routine addresses. In order to translate from VM-driver routine addresses to hypervisor-driver routine addresses the first time, we need to know the correspondence between the original driver's code addresses and the translated driver's code addresses. Although this information can be generated while creating the hypervisor driver from the VM driver, overall, this approach is quite cumbersome.

We reduce the complexity of translating from the VM driver's addresses to the hypervisor driver's addresses by using the same rewritten driver for both the VM driver instance and hypervisor driver instance. For running the rewritten driver as the VM driver, the `stlb` table for the VM driver instance is filled with identify mappings. Thus, the VM driver instance continues to use its original data addresses and functions correctly as before, except that it runs

a little slower.

Using this approach, the code addresses in the VM driver and the hypervisor driver always differ by a constant offset for all routines, and thus address translations between the two can be done in a simple manner.

4.3.2 Loading the hypervisor driver

The rewritten hypervisor driver is loaded into the Xen address space using a modified ELF loader.

During loading, all data references in the hypervisor instance (i.e., the driver's data symbols and the 'imported' Linux data symbols) are resolved to the corresponding symbol addresses of the driver and Linux variables in the dom0 address space. This is done with the help of the module loader in the dom0 kernel, which saves the necessary driver relocation information at the time the original driver is loaded into the dom0 kernel. This ensures that all hypervisor driver data references point only to memory locations in dom0 address space.

Hypervisor driver calls to external driver support routines are also resolved in a special way. Calls to support routines which are implemented in the hypervisor itself are resolved to the hypervisor's implementation. For other driver support routines which are not implemented by the hypervisor, the driver calls are resolved to 'stub' routines in the hypervisor. A separate stub routine is provided for each unimplemented driver support routine. The mapping between the stub routine number and the corresponding support routine in dom0 is saved by the loader. At runtime, when the stub routine is invoked, it initiates a cross-address-space call to the corresponding dom0 routine using the *upcall* mechanism described in section 4.2.2.

The hypervisor needs some additional information for actually invoking the transmit and interrupt handler routines in the hypervisor driver. It needs to know the driver entry points for the transmit and interrupt routines, and also some additional parameters that the driver expects to be passed on each invocation (such as a pointer to the Linux `netdev` structure for the transmit routine). This information is passed to the hypervisor from the dom0 process which

initiates the driver loading.

4.3.3 Invoking the hypervisor driver

The hypervisor transmits and receives packets on behalf of the guest domains by invoking the transmit and interrupt handler routines of the hypervisor driver respectively. The guest domains interface with the hypervisor driver using a new paravirtualized network driver.

For all invocations of the hypervisor driver, all parameters passed to the driver must be valid heap addresses in dom0 address space. Thus, all packet buffers (`sk_buff` structures in Linux) allocated to the hypervisor driver reside in dom0 address space, and are also persistently mapped into hypervisor address space using the `stlb` mapping mechanism.

For transmit operations from guest domains, the hypervisor acquires a pre-allocated `sk_buff` in dom0 address space, copies the header of the guest packet (up to the first 96 bytes) into the `sk_buff` header, and chains together the rest of the guest packet using the page fragment pointers in the `sk_buff` (using pre-allocated page frames from dom0). It then invokes the hypervisor driver transmit routine with the dom0 `sk_buff` parameter.

The DMA transfers set up by the hypervisor driver work correctly because the hypervisor implementation of the DMA mapping functions, `dma_map_single` and `dma_map_page` return the correct guest machine page addresses. [4]

For receive operations, the hypervisor calls the driver interrupt routine on receiving an interrupt from the NIC. Network packets are received by the hypervisor driver into dom0 `sk_buffs` which are persistently mapped into the hypervisor. The hypervisor demultiplexes the received packets based on the destination MAC address, and queues the packet to the appropriate guest domain. When the guest domain is scheduled next, the hypervisor copies the packets into guest domain buffers, and raises a virtual interrupt to notify the guest domain paravirtual driver.

[4] Alternatively, the hypervisor makes use of the *physical_to_machine* mapping table in the dom0 kernel to map from physical page frames of the skb in dom0 to the correct machine page frames in guest domains. This way, the DMA mapping driver functions can be even invoked using upcalls and would still work correctly.

4.4 Evaluation

4.4.1 Experimental setup

We have implemented TwinDrivers in Xen-3.2.1 (changeset 16485) running Linux version 2.6.18.8 in dom0 and in the guest domains. We evaluate the network performance of a guest domain using TwinDrivers, comparing it with the performance of an unoptimized Xen guest domain, Xen dom0, and native Linux. We evaluate the performance for two workloads. Our first workload uses a netperf [NET] like microbenchmark to measure the transmit and receive network performance in guest domains. Our second workload consists of a web server serving concurrent HTTP requests for files from a SPECweb99 [spe] like file-set.

The testbed consists of a 3.0 GHz Intel Xeon server machine equipped with five Intel Pro1000 Gigabit Ethernet cards. This machine is connected to five client machines (3.0 GHz Intel Xeon) equipped with one Intel Pro1000 Gigabit Ethernet card each.

4.4.2 Microbenchmark Results

The microbenchmark workload measures the maximum TCP streaming throughput achievable over a small set of TCP connections. In the experiments (both transmit and receive), the server machine is connected to each client machine using a separate TCP connection over a different NIC. The experiment measures the maximum aggregate TCP throughput (transmit or receive) the server can achieve using all five NICs.

Figures 4.4 and 4.5 show the transmit and receive performance of a Xen guest domain using TwinDrivers ("domU-twin") and compare it with the performance achieved in an unoptimized guest domain using standard Xen networking ("domU"), the driver domain ("dom0"), and a native Linux system ("Linux"). For the domU-twin configuration, all 10 functions required for fast-path operation of the hypervisor driver (see table 4.1) were implemented in the hypervisor, and no upcalls were made.

For the transmit benchmark (figure 4.4), the native Linux system saturates all 5 NICs to

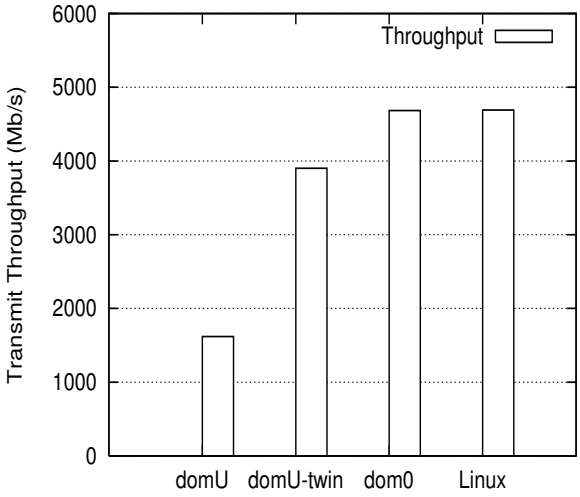

Figure 4.4: Transmit Performance for netperf Benchmark

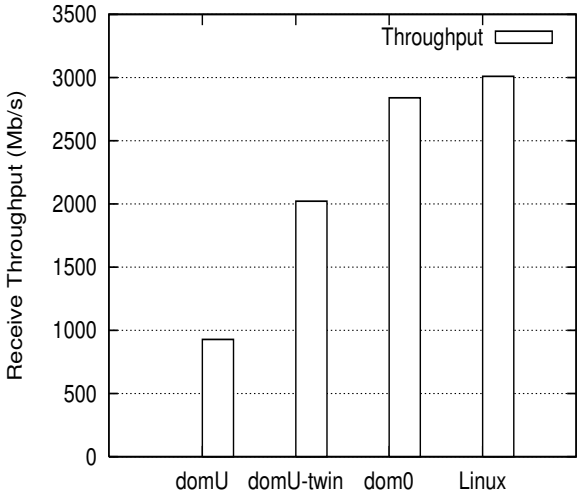

Figure 4.5: Receive Performance for netperf Benchmark

achieve an aggregate throughput of 4690 Mb/s while using only 76.9% of the CPU, while Xen dom0 achieves a throughput of 4683 Mb/s with full CPU saturation. The performance of the TwinDrivers guest domain (domU-twin) is 3902 Mb/s with full CPU saturation, which is within 83% of the dom0 performance, and is within 64% of the native Linux performance, in CPU-scaled units.

The performance of the unoptimized guest domain (domU) itself is only 1619 Mb/s at 100%

CPU saturation. Thus, compared to the unoptimized guest domain, the TwinDriver guest domain achieves a performance improvement of a factor of 2.41.

For the receive benchmark (figure 4.5), the native Linux performance is 3010 Mb/s at full CPU saturation, and the Xen dom0 performance is 2839 Mb/s. The performance of the Twin-Drivers guest domain (domU-twin) is 2022 Mb/s at full CPU saturation, which is roughly 71% of the dom0 performance, and close to 67% of the native Linux performance.

The performance of the unoptimized guest domain is only 928 Mb/s at 100% CPU saturation. Thus, the TwinDrivers guest domain improves upon the guest domain performance by a factor of 2.17.

Figure 4.6 shows the breakdown of packet processing overhead for the transmit workload in the four systems. This profile was obtained with the microbenchmark running only on a single Gigabit NIC. Thus, the relative numbers obtained here differ a little from the throughput results. We show the CPU overhead in terms of cycles per packet incurred in four categories: the dom0 kernel (dom0), the guest domain kernel (domU), the Xen hypervisor (Xen) and the network driver (e1000). For the native Linux case, we show the Linux kernel overhead in the dom0 kernel.

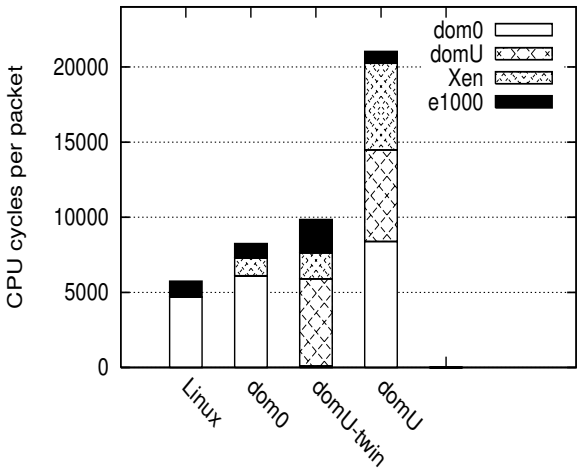

Figure 4.6: CPU cycles per packet for transmit workload

The unoptimized guest domain per-packet overhead is more than twice the overhead of the TwinDriver guest domain (21159 cycles/packet vs. 9972 cycles/packet). Most of this overhead is incurred in invoking dom0 (8394 cycles/packet) and in the additional hypervisor overhead for switching and transferring packets between the guest and driver domain [STJP08]. The TwinDrivers guest avoids both these overheads by directly invoking the hypervisor driver.

Compared to native Linux, both the dom0 and the TwinDrivers guest incur the virtualization overhead of running on top of a hypervisor (1184 cycles/packet for dom0, and 1726 cycles/packet for domU-twin). In the TwinDrivers configuration, there is additional overhead relative to dom0 in two main areas: the overhead of running a rewritten driver instead of native driver (2218 cycles/packet vs. 960 cycles/packet), and the additional hypervisor overhead of the hypercall interface between the paravirtual driver and the hypervisor driver. Overall, the TwinDrivers guest incurs roughly 20% higher overhead than dom0.

Figure 4.7 shows a similar breakdown of the overhead for the receive workload.

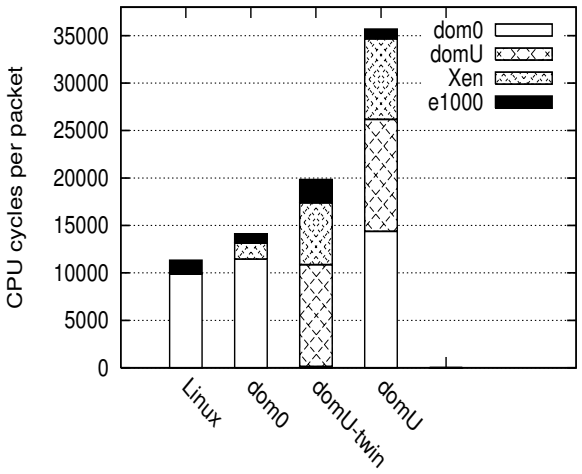

Figure 4.7: CPU cycles per packet for receive workload

Here again, the unoptimized guest domain incurs almost twice the per-packet overhead as the TwinDrivers guest domain (35905 cycles/packet vs. 20089 cycles/packet), and most of this overhead is incurred in invoking dom0 (14384 cycles/packet) and in additional hypervisor

overheads. By invoking the hypervisor driver directly, the TwinDrivers guest avoids most of these overheads.

Compared to dom0 and native Linux performance, the TwinDrivers per-packet overhead is quite large (20089 cycles/packet vs. 14308 and 11166 cycles/packet). Part of this can be explained as the overhead of running the rewritten driver (2445 cycles/packet vs. 972-1422 cycles/packet), but a large part of the TwinDriver receive overhead is incurred in the hypervisor itself (6514 cycles/packet). More detailed profiling shows that most of this overhead (3525 cycles/packet) is incurred in copying the packet from the hypervisor driver to the guest domain driver.

For both the transmit and receive workloads, the overhead incurred by running a binary rewritten driver instead of a native driver is relatively small. The rewritten driver runs slower by a factor of roughly 2 to 3, but, the impact of this slowdown on the overall overhead of the guest domain is relatively low, less than 15%.

4.4.3 Web Server Workload

We now compare the performance of the guest domain using TwinDrivers with the original guest domain, dom0 and a native Linux system, for a web server workload. In this experiment, the server machine runs the *knot* web server, a bare-bones, lightweight web server developed as part of the Capriccio project [vBCZ+03]. It serves a static set of files generated from the file size distribution specified in the static content part of SPWECweb'99 [spe]. Since we are only interested in the network performance, we use a file-set consisting of only a single directory. This entire file-set fits in memory and does not stress the disk I/O subsystem.

The workload for the web server is generated by running httperf [MJ98] on a set of client machines. Requests are generated in an 'open' loop, and responses from the server are discarded if they are not received within a certain timeout.

Figure 4.8 compares the performance of the web server running in the guest domain using TwinDrivers ("domU-twin"), the original guest domain ("domU"), dom0 ("dom0"), and a native

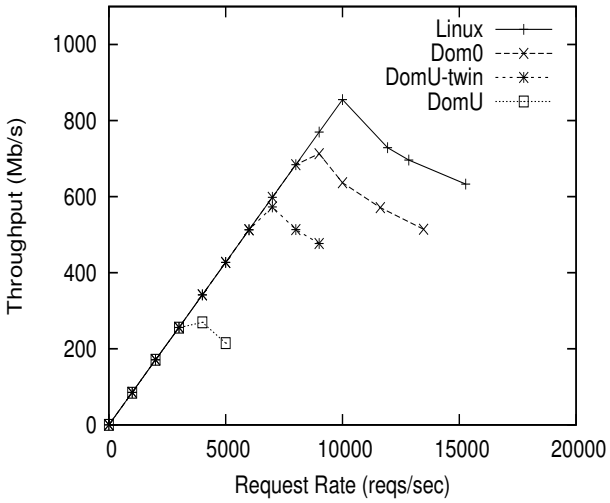

Figure 4.8: Web Server Workload

Linux system ("Linux"). The figure plots the aggregate throughput of responses received by all httperf clients (in Mb/s) as a function of the total connection request rate issued by the clients. In the figure, all configurations could not be tested to the same request rate because some configurations could not sustain high connection rates, and thus effectively ran at a lower connection rate even when a higher rate was requested.

The overall trends seen in the web server workload are similar to the trends seen for the microbenchmarks. The maximum throughput achieved by the native Linux configuration is is 855 Mb/s. dom0 achieves a peak throughput of 712 Mb/s. The original Xen guest domain achieves a peak throughput of 269 Mb/s, which is only 31% of the native Linux performance. The TwinDrivers guest domain achieves a peak throughput of 572 Mb/s, which is a more than factor of 2 improvement over the unoptimized guest domain performance, and is roughly within 67% of native Linux performance.

4.4.4 Cost of Upcalls

To achieve good performance in the hypervisor driver, upcalls to driver support routines must be avoided during the performance critical parts of the driver. Table 4.1 shows that for the

Intel e1000 driver, there are 10 driver support routines that are called on the fast path. Figure 4.9 shows how the transmit performance of a TwinDrivers guest domain drops when not all the necessary upcalls are implemented in the hypervisor.

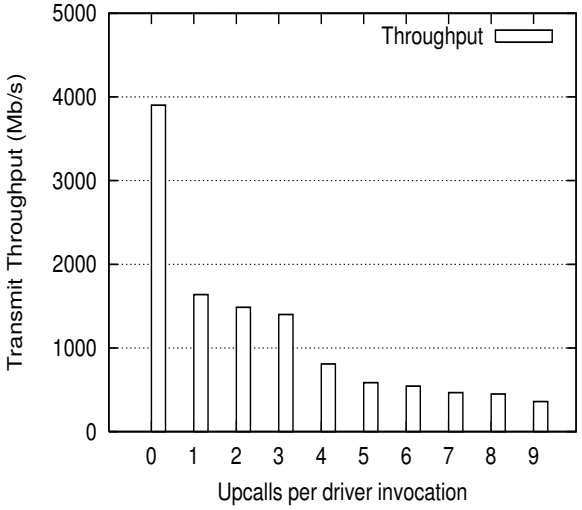

Figure 4.9: Transmit throughput as a function of number of upcalls

The X axis shows the number of performance-critical support routines for which the hypervisor has to make an upcall. When no upcalls are made (first bar), transmit performance is 3902 Mb/s. As soon as the hypervisor has to make even one upcall per driver invocation, the performance drops to 1638 Mb/s (second bar). The performance drops progressively as more and more upcalls are needed, until finally it drops to 359 Mb/s when all but the network receive function (`netif_rx` in Linux) are implemented as upcalls.

4.4.5 Engineering Effort

The 10 driver support routines listed in table 4.1 were implemented in the Xen hypervisor. The entire implementation took 851 lines of commented C code and header files. This is a very small development effort compared to the effort that would be needed to support the entire driver support interface.

4.5 Summary

In this chapter, we presented TwinDrivers, an I/O architecture for virtual machines which combines the performance benefits of hypervisor based VMMs with the safety and software engineering benefits of hosted VMMs. TwinDrivers uses binary rewriting techniques to *automatically* derive safe and efficient hypervisor drivers from the host OS driver. In contrast with the per-packet optimizations discussed in chapter 3, the TwinDrivers approach is more general, because it provides performance improvement for all network workloads, not just single-connection oriented traffic intensive workloads.

Chapter 5

Concurrent Direct Network Access

The optimizations discussed so far were software-only optimizations, meaning that they could be implemented without the need for any specialized hardware support, such as from the NIC or the CPU. Thus these optimizations were applicable across all hardware platforms.

In this chapter, we discuss a network virtualization architecture in which we loosen this restriction and allow modifications to the NIC hardware in order to provide better support for networking in virtual machines. We present the concurrent direct network access (CDNA) architecture, a new I/O virtualization technique combining both software and hardware components for reducing the overhead of network virtualization in Xen.

The CDNA network virtualization architecture provides guest domains running on the Xen VMM with a safe direct access to the network interface. With CDNA, each guest domain is allocated a unique *context* on the network interface and communicates directly with the network interface with that context. In this manner, the guest domains running on Xen operate as if each has access to its own dedicated network interface.

The CDNA network virtualization architecture achieves dramatic increase in network efficiency by dividing the tasks of traffic multiplexing, interrupt delivery and memory protection among hardware and software in a novel way. Traffic multiplexing is performed directly on the network interface, whereas interrupt delivery and memory protection are performed by the Xen

VMM with support from the network interface. This division of tasks into hardware and software components simplifies the overall software architecture, minimizes the hardware additions to the network interface, and addresses the network performance bottlenecks.

We describe the design of the CDNA architecture in section 5.1, and its implementation on an FPGA based network interface card in section 5.2. We evaluate the performance benefits of the CDNA architecture in section 5.3.

5.1 Concurrent Direct Network Access

In CDNA, the network interface and the hypervisor collaborate to provide the abstraction that each guest domain is connected directly to its own network interface. Figure 5.1 shows the CDNA architecture. The network interface must support multiple *contexts* in hardware. Each context acts as if it is an independent physical network interface and can be controlled by a separate device driver instance. Instead of assigning ownership of the entire network interface to the driver domain, the hypervisor treats each context as if it were a physical NIC and assigns ownership of contexts to guest domains. Notice the absence of the driver domain from the figure: each guest can transmit and receive network traffic using its own private context without any interaction with other guest domains or the driver domain. The driver domain, however, is still present to perform control functions and allow access to other I/O devices. Furthermore, the hypervisor is still involved in networking, as it must guarantee memory protection and deliver virtual interrupts to the guest domains.

With CDNA, the communication overheads between guest and driver domains and the software multiplexing overheads within the driver domain are eliminated entirely. However, the network interface must now multiplex the traffic across all of its active contexts. The following subsections describe how CDNA performs traffic multiplexing, interrupt delivery and DMA memory protection.

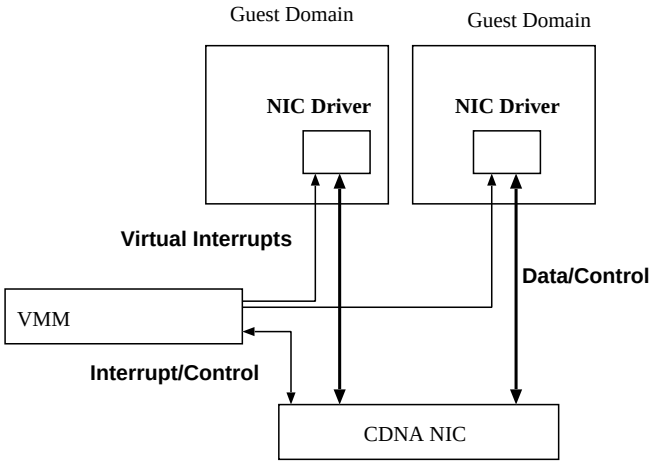

Figure 5.1: CDNA Architecture. Each guest domain running on Xen is given direct access to its own private virtual context on the CDNA NIC.

5.1.1 Multiplexing Network Traffic

CDNA eliminates the software multiplexing overheads within the driver domain by multiplexing network traffic on the NIC. The network interface must be able to identify the source or target guest domain for all network traffic. The NIC accomplishes this by providing independent hardware contexts and associating a unique Ethernet MAC address with each context. The hypervisor assigns a unique hardware context on the NIC to each guest domain. The device driver within the guest domain then interacts with its context exactly as if the context were an independent physical network interface. As in a typical driver, these interactions consist of creating DMA descriptors and updating a mailbox on the NIC via PIO.

Each context on the network interface therefore must include a unique set of mailboxes. This isolates the activity of each guest domain, so that the NIC can distinguish between the different guests. The hypervisor assigns a context to a guest simply by mapping the I/O locations for that context's mailboxes into the guest's address space. The hypervisor also notifies the NIC that the context has been allocated and is active. As the hypervisor only maps each context into a single guest's address space, a guest cannot accidentally or intentionally access any context on the NIC other than its own. When necessary, the hypervisor can also revoke a context at

any time by notifying the NIC, which will shut down all pending operations associated with the indicated context.

To multiplex transmit network traffic, the NIC simply services all of the hardware contexts fairly and interleaves the network traffic for each guest. When network packets are received by the NIC, it uses the Ethernet MAC address to demultiplex the traffic, and transfers each packet to the appropriate guest using available DMA descriptors from that guest's context.

5.1.2 Interrupt Delivery

In addition to isolating the guest domains and multiplexing network traffic, the hardware contexts on the NIC must also be able to interrupt their respective guests. As the NIC carries out network requests on behalf of any particular context, the CDNA NIC updates that context's consumer pointers for the DMA descriptor rings. Normally, the NIC would then interrupt the guest to notify it that the context state has changed. However, in Xen all physical interrupts are handled by the hypervisor. Therefore, the NIC cannot physically interrupt the guest domains directly. Even if it were possible to interrupt the guests directly, that could create a much higher interrupt load on the system, which would decrease the performance benefits of CDNA.

Under CDNA, the NIC keeps track of which contexts have been updated since the last physical interrupt, encoding this set of contexts in an interrupt bit vector. The NIC transfers an interrupt bit vector into the hypervisor's memory space using DMA. The interrupt bit vectors are stored in a circular buffer using a producer/consumer protocol to ensure that they are processed by the host before being overwritten by the NIC. After an interrupt bit vector is transferred, the NIC raises a physical interrupt, which invokes the hypervisor's interrupt service routine. The hypervisor then decodes all of the pending interrupt bit vectors and schedules virtual interrupts to each of the guest domains that have pending updates from the NIC. When the guest domains are next scheduled by the hypervisor, the CDNA network interface driver within the guest receives these virtual interrupts as if they were actual physical interrupts from the hardware. At that time, the driver examines the updates from the NIC and determines

what further action, such as processing received packets, is required.

5.1.3 DMA Memory Protection

In the x86 architecture, network interfaces and other I/O devices use physical addresses when reading or writing host system memory. The device driver in the host operating system is responsible for doing virtual-to-physical address translation for the device. The physical addresses are provided to the network interface through read and write DMA descriptors. By exposing physical addresses to the network interface, the DMA engine on the NIC can be co-opted into compromising system security by a buggy or malicious driver. There are two key I/O protection violations that are possible in the x86 architecture. First, the device driver could instruct the NIC to transmit packets containing a payload from physical memory that does not contain packets generated by the operating system, thereby creating a security hole. Second, the device driver could instruct the NIC to receive packets into physical memory that was not designated as an available receive buffer, possibly corrupting memory that is in use.

In the conventional Xen network architecture discussed in chapter 2, Xen trusts the device driver in the driver domain to only use the physical addresses of network buffers in the driver domain's address space when passing DMA descriptors to the network interface. This ensures that all network traffic will be transferred to/from network buffers within the driver domain. Since guest domains do not interact with the NIC, they cannot initiate DMA operations, so they are prevented from causing either of the I/O protection violations in the x86 architecture.

This solution is insufficient for the CDNA architecture. In a CDNA system, device drivers in the guest domains have direct access to the network interface and are able to pass DMA descriptors with physical addresses to the device. Thus, the untrusted guests could read or write memory in any other domain through the NIC, unless additional security features are added. To maintain isolation between guests, the CDNA architecture validates and protects all DMA descriptors and ensures that a guest maintains ownership of physical pages that are sources or targets of outstanding DMA accesses. Although the hypervisor and the network interface share

the responsibility for implementing these protection mechanisms, the more complex aspects are implemented in the hypervisor.

The most important protection provided by CDNA is that it does not allow guest domains to directly enqueue DMA descriptors into the network interface descriptor rings. Instead, the device driver in each guest must call into the hypervisor to perform the enqueue operation. This allows the hypervisor to validate that the physical addresses provided by the guest are, in fact, owned by that guest domain. This prevents a guest domain from arbitrarily transmitting from or receiving into another guest domain. The hypervisor prevents guest operating systems from independently enqueueing unauthorized DMA descriptors by establishing the hypervisor's exclusive write access to the host memory region containing the CDNA descriptor rings during driver initialization.

5.1.4 Discussion

The handling of the DMA descriptors within the hypervisor is linked to a particular network interface only because the format of the DMA descriptors and their rings is likely to be different for each device. As the hypervisor must validate that the host addresses referred to in each descriptor belong to the guest operating system that provided them, the hypervisor must be aware of the descriptor format. Fortunately, there are only three fields of interest in any DMA descriptor: an address, a length, and additional flags. This commonality should make it possible to generalize the mechanisms within the hypervisor by having the NIC notify the hypervisor of its preferred format. The NIC would only need to specify the size of the descriptor and the location of the address, length, and flags. The hypervisor would not need to interpret the flags, so they could just be copied into the appropriate location.

CDNA's DMA memory protection is specific to Xen only insofar as Xen permits guest domains to use physical memory addresses. Consequently, the current implementation must validate the ownership of those physical addresses for every requested DMA operation. For VMMs that only permit the guest to use virtual addresses, the hypervisor could just as easily

translate those virtual addresses and ensure physical contiguity. The current CDNA implementation does not rely on physical addresses in the guest at all; rather, a small library translates the driver's virtual addresses to physical addresses within the guest's driver before making a hypercall request to enqueue a DMA descriptor. For VMMs that use virtual addresses, this library would do nothing.

5.2 CDNA NIC Implementation

To evaluate the CDNA concept in a real system, RiceNIC, a programmable and reconfigurable FPGA-based Gigabit Ethernet network interface [SR06], was modified to provide virtualization support. RiceNIC contains a Virtex- II Pro FPGA with two embedded 300MHz PowerPC processors, hundreds of megabytes of on-board SRAM and DRAM memories, a Gigabit Ethernet PHY, and a 64- bit/66 MHz PCI interface [ADS03]. Custom hardware assist units for accelerated DMA transfers and MAC packet handling are provided on the FPGA. The RiceNIC architecture is similar to the architecture of a conventional network interface. With basic firmware and the appropriate Linux or FreeBSD device driver, it acts as a standard Gigabit Ethernet network interface that is capable of fully saturating the Ethernet link while only using one of the two embedded processors.

To support CDNA, both the hardware and firmware of the RiceNIC were modified to provide multiple protected contexts and to multiplex network traffic. The network interface was also modified to interact with the hypervisor through a dedicated context to allow privileged management operations. The modified hardware and firmware components work together to implement the CDNA interfaces.

To support CDNA, the most significant addition to the network interface is the specialized use of the 2 MB SRAM on the NIC. This SRAM is accessible via PIO from the host. For CDNA, 128 KB of the SRAM is divided into 32 partitions of 4 KB each. Each of these partitions is an interface to a separate hardware context on the NIC. Only the SRAM can be memory mapped into the host's address space, so no other memory locations on the NIC are accessible via PIO.

As a context's memory partition is the same size as a page on the host system and because the region is page-aligned, the hypervisor can trivially map each context into a different guest domain's address space. The device drivers in the guest domains may use these 4 KB partitions as general purpose shared memory between the corresponding guest operating system and the network interface.

Within each context's partition, the lowest 24 memory locations are mailboxes that can be used to communicate from the driver to the NIC. When any mailbox is written by PIO, a global mailbox event is automatically generated by the FPGA hardware. The NIC firmware can then process the event and efficiently determine which mailbox and corresponding context has been written by decoding a two level hierarchy of bit vectors. All of the bit vectors are generated automatically by the hardware and stored in a data scratchpad for high speed access by the processor. The first bit vector in the hierarchy determines which of the 32 potential contexts have updated mailbox events to process, and the second vector in the hierarchy determines which mailbox(es) in a particular context have been updated. Once the specific mailbox has been identified, that off-chip SRAM location can be read by the firmware and the mailbox information processed.

Each context requires 128 KB of storage on the NIC for metadata, such as the rings of transmit- and receive- DMA descriptors provided by the host operating systems. Furthermore, each context uses 128 KB of memory on the NIC for buffering transmit packet data and 128 KB for receive packet data. However, the NIC's transmit and receive packet buffers are each managed globally, and hence packet buffering is shared across all contexts.

The modifications to the RiceNIC to support CDNA were minimal. The major hardware change was the additional mailbox storage and handling logic. This could easily be added to an existing NIC without interfering with the normal operation of the network interface – unvirtualized device drivers would use a single context's mailboxes to interact with the base firmware. Furthermore, the computation and storage requirements of CDNA are minimal. Only one of the RiceNIC s two embedded processors is needed to saturate the network, and

only 12 MB of memory on the NIC is needed to support 32 contexts. Therefore, with minor modifications, commodity network interfaces could easily provide sufficient computation and storage resources to support CDNA.

5.3 Evaluation

5.3.1 Experimental Setup

The performance of Xen and CDNA network virtualization was evaluated on an AMD Opteron-based system running Xen 3 Unstable. This system used a Tyan S2882 motherboard with a single Opteron 250 processor and 4GB of DDR400 SDRAM. Xen 3 Unstable was used because it provides the latest support for high-performance networking, including TCP segmentation offloading, and the most recent version of Xenoprof [MST+05] for profiling the entire system.

In all experiments, the driver domain was configured with 256MB of memory and each of 24 guest domains were configured with 128MB of memory. Each guest domain ran a stripped-down Linux 2.6.16.29 kernel with minimal services for memory efficiency and performance. For the base Xen experiments, a single dual-port Intel Pro/1000MT NIC was used in the system. In the CDNA experiments, two RiceNICs configured to support CDNA were used in the system. Linux TCP parameters and NIC coalescing options were tuned in the driver domain and guest domains for optimal performance. For all experiments, checksum offloading and scatter/gather I/O were enabled. TCP segmentation offloading was enabled for experiments using the Intel NICs, but disabled for those using the RiceNICs due to lack of support. The Xen system was setup to communicate with a similar Opteron system that was running a native Linux kernel. This system was tuned so that it could easily saturate two NICs both transmitting and receiving so that it would never be the bottleneck in any of the tests.

To validate the performance of the CDNA approach, multiple simultaneous connections across multiple NICs to multiple guests domains were needed. A multithreaded, event-driven, lightweight network benchmark program was developed to distribute traffic across a configurable

number of connections. The benchmark program balances the bandwidth across all connections to ensure fairness and uses a single buffer per thread to send and receive data to minimize the memory footprint and improve cache performance.

5.3.2 Single Guest Performance

Tables 5.1 and 5.2 show the transmit and receive performance of a single guest operating system over two physical network interfaces using Xen and CDNA. The first two rows of each table show the performance of the Xen I/O virtualization architecture using both the Intel and RiceNIC network interfaces. The third row of each table shows the performance of the CDNA I/O virtualization architecture.

System	NIC	Mb/s	Hyp	Driver domain	Guest domain	Idle
Xen	Intel	1602	19.8%	35.7%	39.7%	3.0%
Xen	RiceNIC	1674	13.7%	41.5%	39.5%	3.8%
CDNA	RiceNIC	1867	10.2%	0.3%	37.8%	50.8%

Table 5.1: Comparison of transmit performance for a single Xen guest domain with 2 NICs using the Xen I/O architecture and the CDNA architecture

System	NIC	Mb/s	Hyp	Driver domain	Guest domain	Idle
Xen	Intel	1112	25.7%	36.8%	31.0%	5.0%
Xen	RiceNIC	1075	30.6%	39.4%	28.8%	0%
CDNA	RiceNIC	1874	9.9%	0.3%	48.0%	40.9%

Table 5.2: Comparison of receive performance for a single Xen guest domain with 2 NICs using the Xen I/O architecture and the CDNA architecture

The Intel network interface is used with Xen only through the use of software virtualization. However, the RiceNIC can be used with both CDNA and software virtualization. To use the RiceNIC interface with software virtualization, a context was assigned to the driver domain and no contexts were assigned to the guest operating system. Therefore, all network traffic from the guest operating system is routed via the driver domain as it normally would be, through the use of software virtualization. Within the driver domain, all of the mechanisms within the CDNA NIC are used identically to the way they would be used by a guest domain when configured to

use concurrent direct network access. As the tables show, the Intel network interface performs similarly to the RiceNIC network interface. Therefore, the benefits achieved with CDNA are the result of the CDNA I/O virtualization architecture, not the result of differences in network interface performance.

Table 5.1 shows that using all of the available processing resources, Xen's software virtualization is not able to transmit at line rate over two network interfaces with either the Intel hardware or the RiceNIC hardware. However, only 41% of the processor is used by the guest domain. The remaining resources are consumed by Xen overheads using the Intel hardware, approximately 20% in the hypervisor and 37% in the driver domain performing software multiplexing and other tasks.

As the table shows, CDNA is able to saturate two network interfaces, whereas traditional Xen networking cannot. Additionally, CDNA performs far more efficiently, with 51% processor idle time. The increase in idle time is primarily the result of two factors. First, nearly all of the time spent in the driver domain is eliminated. The remaining time spent in the driver domain is unrelated to networking tasks. Second, the time spent in the hypervisor is decreased. With Xen, the hypervisor spends the bulk of its time managing the interactions between the front-end and back-end virtual network interface drivers. CDNA eliminates these communication overheads with the driver domain, so the hypervisor instead spends the bulk of its time managing DMA memory protection.

Table 5.2 shows the receive performance of the same configurations. Receiving network traffic requires more processor resources, so Xen only achieves 1112 Mb/s with the Intel network interface, and slightly lower with the RiceNIC interface. Again, Xen overheads consume the bulk of the time, as the guest domain only consumes about 32% of the processor resources when using the Intel hardware.

In summary, the CDNA I/O virtualization architecture provides significant performance improvements over Xen for both transmit and receive. On the transmit side, CDNA requires half the processor resources to deliver about 200 Mb/s higher throughput. On the receive side,

CDNA requires 60% of the processor resources to deliver about 750 Mb/s higher throughput.

5.3.3 Memory Protection

The software-based protection mechanisms in CDNA can potentially be replaced by a hardware IOMMU. For example, AMD has proposed an IOMMU architecture for virtualization that restricts the physical memory that can be accessed by each device [AMD05]. AMD's proposed architecture provides memory protection as long as each device is only accessed by a single domain. For CDNA, such an IOMMU would have to be extended to work on a per-context basis, rather than a per-device basis. This would also require a mechanism to indicate a context for each DMA transfer. Since CDNA only distinguishes between guest domains and not traffic flows, there are a limited number of contexts, which may make a generic system-level context aware IOMMU practical.

System	DMA Protection	Mb/s	Hyp	Driver domain	Guest domain	Idle
CDNA (Transmit)	Enabled	1867	10.2%	0.3%	37.8%	50.8%
CDNA (Transmit)	Disabled	1867	1.9%	0.2%	37.0%	60.4%
CDNA (Receive)	Enabled	1874	9.9%	0.3%	48.0%	40.9%
CDNA (Receive)	Disabled	1874	1.9%	0.2%	47.2%	50.2%

Table 5.3: CDNA 2-NIC Transmit and Receive performance with and without DMA protection

Table 5.3 shows the performance of the CDNA I/O virtualization architecture both with and without DMA memory protection. (The performance of CDNA with DMA memory protection enabled was replicated from Tables 5.1 and 5.2 for comparison purposes.) By disabling DMA memory protection, the performance of the modified CDNA system establishes an upper bound on achievable performance in a system with an appropriate IOMMU. However, there would be additional hypervisor overhead to manage the IOMMU that is not accounted for by this experiment. Since CDNA can already saturate two network interfaces for both transmit and receive traffic, the effect of removing DMA protection is to increase the idle time by about 9%. As the table shows, this increase in idle time is the direct result of reducing the number of hypercalls from the guests and the time spent in the hypervisor performing protection operations.

5.4 Discussion

The CDNA architecture and the TwinDrivers architecture represent two different approaches to solving the problem of optimizing network performance in a virtual machine environment. While the TwinDrivers architecture takes a software approach towards eliminating the switching overheads of a hosted VMM architecture, the CDNA approach tackles the same problem by modifying the NIC hardware to eliminate the driver domain from the performance critical path.

The relative merits of using a hardware vs. a software approach to solving the problem of network performance are debatable. The CDNA approach to network interface virtualization may be slightly more efficient than the software TwinDrivers approach, especially for receive workloads where CDNA DMAs the received packets directly into guest memory and avoids the double data copy inherent in the TwinDrivers approach. On the other hand, using a software approach like TwinDrivers allows us to completely decouple the guest domain from any hardware dependency. This allows for greater flexibility in the management of VMs for operations such as VM migration. More importantly, the benefits of the CDNA approach are tied to using a particular network interface, whereas the TwinDrivers approach provides performance benefits irrespective of the underlying hardware.

Chapter 6

Related Work

This thesis explores the space of software and hardware techniques for optimizing network I/O performance in virtual machines. As such, it relates to research work in the areas of VMMs, network optimizations, device drivers and user-level networking.

6.1 Virtual machine monitors

Virtualization first became available with the IBM VM/370 [Gum83, SM79] to allow legacy code to run on new hardware platform. Currently, a number of software vendors provide x86 virtualization solutions, such as VMware, XenSource (Citrix) and Microsoft Hyper-V. Several studies have documented the cost of full virtualization, especially on architectures such as the Intel x86 [AA06]. To address these performance problems, paravirtualization has been introduced, with Xen [BDF+03, FHN+04] as its most popular representative. Denali [WSG02] similarly attempts to support a large number of virtual machines.

The use of driver domains to host device drivers has become popular for reasons of reliability and extensibility. Examples include the Xen 2.0 architecture and VMware's hosted workstation [SVL01]. The downside of this approach is a performance penalty for device access, documented, among others, in Sugerman et al. [SVL01] and in Menon et al. [MST+05]. Sugerman et al. describe the VMware hosted virtual machine monitor in [SVL01], and describe the major

sources of network virtualization overhead in this architecture. In a fully virtualized system, 'world switches' incurred for emulating I/O access to virtual devices are the biggest source of overhead.

King et al. [KDC03] describe optimizations to improve overall system performance for Type-II virtual machines. Their optimizations include reducing context switches to the VMM process by moving the VM support into the host kernel, reducing memory protection overhead by using segment bounds, and reducing context switch overhead by supporting multiple addresses spaces within a single process.

6.2 Networking optimizations

There is a huge body of research in the networking literature focusing on analysis and optimization of the network stack. Initial analysis of TCP performance [CJRS89] identified the per-byte data touching operations to be the major source of overhead for TCP. This led to the development of a number of techniques for avoiding data copy, both in software [PDZ00] [KC96], and hardware [MRK+03, KSZ95]. Techniques such as zero-copy transmit and hardware checksum offload have now become common in modern network cards [MI04, Cor03, FHH+03].

Later work [KP96] identified the per-packet overhead as the dominant source of overhead for real-world workloads, which are dominated by small message sizes. This led to the development of offloading techniques for reducing per-packet overheads, such as TCP segmentation offload.

Recently, some high end network cards have started providing more complex offload support for TCP receive processing, such as Large Receive Offload (LRO) in Neterion NICs [Gro05]. The idea of LRO is similar to that of Receive Aggregation, except that it is performed in the NIC, and thus it can reduce the per-packet overhead incurred in the network driver. However, a pure-software approach such as Receive Aggregation is much more generic, and can yield much of the benefit of packet aggregation in a hardware independent manner. Additionally, the Neterion NIC does not support Acknowledgment Offload, and thus does not offer support for reducing the overhead on the ACK transmit path.

Jumbo frames, which allow the Ethernet MTU size to be set to 9000 bytes, can also effectively help reduce the per-packet overheads for bulk data transfers. However, they require the whole LAN network to be upgraded to use the same MTU size. Receive Aggregation and Acknowledgment Offload are effective at improving the network stack performance irrespective of the network MTU size or networking hardware used.

6.3 Device driver safety and reuse

Device drivers have received an enormous amount of interest from the research community. The TwinDrivers architecture builds on the notion of running device drivers in a virtual machine [FHN+04, LUSG04], but we go beyond that work in allowing safe execution of the performance-critical parts of the driver in the hypervisor. The idea of executing device drivers in user-level processes is similar, and we project that it can benefit from our techniques as well [HBG+06].

Reusing drivers developed for one environment in a new environment is a difficult task. In the Flux OSkit [FBB+97], driver sources are ported from the original OS into a new OS by re-implementing the entire driver-kernel API support library. The kernel driver support library is a large and poorly documented body of code. Reimplementing this library requires a deep understanding of the internals of the original OS, and can be a a source of subtle bugs. The issues arising from the semantic differences between the old and new OS environments are discussed in detail in [LUSG04].

The VMware ESX server [Wal02] runs selected drivers in the hypervisor by porting the drivers and their support routines to the VMM. Not only does this involve significant development effort, it also leaves the hypervisor vulnerable to bugs in the driver. Our work ensures that the rewritten hypervisor driver executes safely in the hypervisor. Additionally, since the number of driver support routines needed to run the error-free performance-critical path in the driver is very small, the software development costs of our approach are significantly smaller.

Numerous attempts have been made to reduce the vulnerability of the kernel to device driver crashes, without going all the way to running the device driver in a virtual machine

[SBL03, WLAG93]. These approaches typically constrain memory accesses by the drivers to prevent wild writes that corrupt the kernel data structures, either by erecting address space barriers or by checking memory accesses. In our approach, we avoid any vulnerability of this nature as a by-product of leaving all the driver data structures in the virtual machine and not allowing the driver any access to the hypervisor data structures.

A number of research efforts have looked at mechanisms to safely extend operating system functionality with third-party extensions [BSS+95, SESS96]. The SPIN extensible kernel [BSS+95] guarantees safe execution of extension code by requiring that the kernel and all extensions be written in a type-safe language (Modula-3). In contrast, the TwinDrivers approach does not requires extensions to be written in any particular language, and works with existing compiled driver binaries. The VINO extensible kernel [SESS96] uses software fault isolation [WLAG93] as its safety mechanism. It does not require a translation mechanism such as SVM, because the extension executes in the same address space as the kernel. In contrast, TwinDrivers uses SVM to implement both a protection and a translation mechanism, and this is required because the device driver data is located in an address space that is different from the hypervisor address space.

We borrow from the Microdrivers project [GRB+08] the idea of running performance-critical parts of the driver in the kernel/hypervisor and other parts in user-space processes or in a virtual machine. Many differences, however, exist between the two approaches. First, our hypervisor instance cannot corrupt the hypervisor data structures, while the part of the Microdrivers that runs in the kernel has the potential of crashing the kernel. Methods like those used in SFI [WLAG93] or Nooks [SBL03] have to be used to reduce this vulnerability, potentially leading to extra performance overhead. Second, the Microdrivers approach requires manual annotations for all kernel and driver data structures that can be shared between the user-space and kernel-space driver. These annotations are necessary because Microdrivers use explicit data marshaling to keep the (separate) data structures of the kernel and user driver consistent with each other. In contrast, we use a single copy of all data structures mapped at different virtual addresses

in the hypervisor and the VM, providing us trivially with consistency and obviating the need for marshaling and annotations. Thus, no driver-specific knowledge or engineering effort is required in out approach; our framework works with unmodified binary drivers. Third, unlike Microdrivers we do not *split* the driver. Both instances of our driver are complete, and we can choose what instance to use for what aspects of the functionality of the driver. This allows us, for instance, to leave all of the support functionality for the error handling code in the transmit and receive parts of the network driver out of the hypervisor.

6.4 User-level networking

The CDNA architecture is similar to that of user-level networking architectures that allow processes to bypass the operating system and access the NIC directly [DRM+98, PF01, vEBBV95, vEW98]. Like CDNA, these architectures require DMA memory protection, an interrupt delivery mechanism, and network traffic multiplexing. Both user-level networking architectures and CDNA handle traffic multiplexing on the network interface. The only difference is that user-level NICs handle flows on a per-application basis, whereas CDNA deals with flows on a per-OS basis. However, as the networking software in the operating system is quite different than that for user-level networking, CDNA relies on different mechanisms to implement DMA memory protection and interrupt delivery.

To provide DMA memory protection, user-level networking architectures rely on memory registration with both the operating system and the network interface hardware. CDNA provides DMA memory protection without actively registering buffers on the NIC. Instead, CDNA relies on the hypervisor to enqueue validated buffers to the NIC by augmenting the hypervisor's existing memory-ownership functionality. This avoids costly runtime registration I/O and permits safe DMA operations to and from arbitrary physical addresses.

Chapter 7

Conclusion

In this thesis, we studied the problem of efficiently virtualizing the network interface in Type-II virtual machine monitors, using the Xen VMM as an example of this architecture. We identified the fundamental performance bottlenecks in the network virtualization architecture of Type-II VMMs, and proposed a number of solutions to address these problems. Although the solutions presented in this thesis were implemented and evaluated in the Xen VMM, they are general enough to be applicable to other VMMs which use the same I/O architecture, such as Microsoft Hyper-V.

We showed that locating the device drivers in a separate host VM is the primary reason for performance degradation in Type-II VMMs. This is because of two reasons: 1) switching overheads are incurred for every transition between the host OS and the guest VMs for invoking the device driver in the host OS, and, 2) the overhead incurred by the I/O virtualization operations required to transfer packets between the guest and the host address spaces. In the Xen VMM, these overheads degrade network performance in guest domains by a factor of five relative to the native performance. We presented a detailed analysis of the virtualization overheads incurred in the Xen network virtualization stack, and proposed three solutions to address these performance bottlenecks. The three solutions explore the networking performance that can be achieved while performing network virtualization in the host OS, in the VMM, and

in the NIC hardware.

Our first solution consists of set of packet aggregation optimizations that can be applied without modifying the existing Type-II I/O architecture of the Xen VMM. This solution retains the core functionality of I/O virtualization, including device driver execution, in the Xen driver domain. The main idea of these optimizations was to reduce network virtualization overheads in the Xen I/O stack by using packet 'coalescing' techniques to reduce the per-packet processing overheads. Using these optimizations, we demonstrated performance improvements by a factor of 4 for TCP transmit workloads running in Xen guest domains, and by a factor of 2 for TCP receive workloads.

In our second solution, we proposed a new I/O virtualization architecture, called Twin-Drivers, which moved the task of I/O virtualization and device driver execution from the host OS into the Xen hypervisor. This architecture thus combines the performance advantages of Type-I VMMs with the safety and software engineering benefits of Type-II VMMs. The Twin-Drivers approach makes use of binary rewriting techniques to derive *safe* hypervisor drivers automatically from host OS drivers, and runs these directly in the hypervisor to achieve good performance. The TwinDrivers approach improves networking performance in Xen guest domains by another factor of 2.

Finally, in our third solution, we proposed a hardware based approach for improving network performance in Type-II VMMs. This work was done in collaboration with researchers at Rice University [WSC+07]. The basic idea of this work was to develop a specialized CDNA network interface which supported direct, concurrent access from Xen guest domains to the NIC. The CDNA architecture moves the task of network virtualization into the NIC, and provides each guest domain with a virtual NIC context which it can access directly. Thus, by bypassing the driver domain for network I/O operations, the CDNA architecture achieves performance improvements similar to a Type-I VMM.

Overall, our solutions help significantly bridge the gap between the network performance in a virtualized environment and native environment, eventually achieving within 70% of the

native performance.

Bibliography

[AA06] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS*, 2006.

[ADS03] Avnet Design Services. Xilinx Virtex-II Pro Development Kit: User's Guide, November 2003.

[AMD] AMD64 Architecture Programmer's Manual, Volume 2. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs%/24593.pdf`.

[AMD05] Advanced Micro Devices. Secure Virtual Machine Architecture Reference Manual, May 2005. Revision 3.01.

[ATS] PCI-SIG - ATS Specifications. `http://www.pcisig.com/specifications/iov/ats`.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[BSS+95] Brian N. Bershad, Stefan Savage, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN operating System. In *Symposium on Operating System Principles (SOSP)*, 1995.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6), June 1989.

[Cor03] Intel Corporation. Small packet traffic performance optimization for 8255x and 8254x Ethernet Controllers. Technical Report application Note (AP-453), Sept 2003.

[DRM+98] D. Dunning, G. Reiner, G. McAlpine, D. Cameron, B. Schubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE MIcro*, 18(2), 1998.

[EAV+06] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Operating Systems Design and Implementation (OSDI)*, 2006.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *16th ACM Symposium on Operating System Principles (SOSP'97)*, Saint-Malo, France, October 1997.

[FHH+03] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. Tcp performance re-visited. In *International Symposium on Performance Analysis of Systems and Software, IPASS*, Austin, TX, 2003.

[FHN+04] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Oct 2004.

[GRB+08] Vinod Ganapathy, Matthew Renzelmann, Arini Balakrishnan, Michael Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *13th Inter-*

nantional Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08), Seattle, WA, March 2008.

[Gro05] Leonid Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Ottawa Linux Symposium*, Ottawa, 2005.

[Gum83] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *Operating System Review*, 40(3):80–89, 2006.

[INT] Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/products/processor/manuals`.

[KC96] H. Keng and J. Chu. Zero-copy TCP in Solaris. In *USENIX 1996 Annual Technical Conference*, 1996.

[KDC03] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *USENIX Annual Technical Conference*, Jun 2003.

[KP96] Jonathan Kay and Joseph Pasquale. Profiling and Reducing Processing Overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.

[KSZ95] K. Kleinpaste, P. Steenkiste, and B. Zill. Software Support for Outboard Buffering and Checksumming. In *ACM SIGCOMM Symposium*, 1995.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gotz. Unmodified Driver Reuse and Improved System Dependability via Virtual Machines. In *Operating Systems Design and Implementation (OSDI'04)*, San Francisco, CA, December 2004.

[MCZ06] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In *USENIX Annual Technical Conference*, Boston, MA, June 2006.

[MI04] Srihari Makineni and Ravi Iyer. Architectural characterization of TCP/IP packet processing on the Pentium M microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[MJ98] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998.

[MRI] PCI-SIG - Multi Root IOV. `http://www.pcisig.com/specifications/iov/multi_root`.

[MRK+03] Dave Minturn, Greg Regnier, Jon Krueger, Ravishankar Iyer, and Srihari Makineni. Addressing TCP/IP Processing Challenges Using the IA and IXP Processors. *Intel Technology Journal*, November 2003.

[MST+05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, Chicago, USA, June 2005.

[MZ08] Aravind Menon and Willy Zwaenepoel. Optimizing TCP Receive Performance. In *USENIX Annual Technical Conference*, 2008.

[NET] The netperf benchmark. `http://www.netperf.org/netperf/NetperfPage.html`.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

[PF01] Ian Pratt and Keir Fraser. Arsenic: a user-accessible Gigabit Ethernet interface. In *IEEE INFOCOMM*, pages 67–76, April 2001.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *19th ACM Symposium on Operating System Principles (SOSP'03)*, Bolton Landing, NY, October 2003.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Operating Systems Design and Implementation (OSDI)*, 1996.

[SM79] L. Seawright and R. MacKinnon. Vm/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, pages 44–55, 1979.

[spe] Specweb'99 benchmark. `http://spec.org/web99`.

[SR06] Jeff Schafer and Scott Rixner. A Reconfigurable and Programmable Gigabit Ethernet Network Interface Card. Technical Report TREE0611, Rice University, Department of Electrical and Computer Engineering, December 2006.

[SRI] PCI-SIG - Single Root IOV. `http://www.pcisig.com/specifications/iov/single_root`.

[STJP08] Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Ian Pratt. Bridging the gap between hardware and software techniques for i/o virtualization. In *USENIX Annual Technical Conference*, 2008.

[SVL01] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, Jun 2001.

[vBCZ+03] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Symposium on Operating System Principles*, December 1995.

[vEW98] T. von Eicken and V. Wogels. Evolution of the Virtual Interface Architecture. *Computer*, 31(11), 1998.

[Wal02] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI'02*, 2002.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP'93*, 1993.

[WSC+07] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *HPCA*, 2007.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Operating Systems Design and Implementation*, 2002.

Curriculum Vitae

Aravind Menon received his Bachelors degree in Computer Science and Engineering from the Indian Institute of Technology (IIT), Kanpur in 2002. He joined the Pre-Doctoral School in Computer and Communication Sciences at EPFL in 2003, and joined the Laboratory of Operating Systems (LABOS) as a Ph.D. student in 2004. His advisor is Prof. Willy Zwaenepoel.

The work done in this thesis resulted in the following publications:

[MST03] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, Chicago, USA, June 2005.

[MCZ06] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In *USENIX Annual Technical Conference*, Boston, MA, June 2006.

[WSC07] Paul Willmann, Jeffreu Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *HPCA*, 2007.

[MZ08] Aravind Menon and Willy Zwaenepoel. Optimizing TCP Receive Performance. In *USENIX Annual Technical Conference*, 2008.

[MSZ09] Aravind Menon, Simon Schubert, and Willy Zwaenepoel. TwinDrivers: Automatic Derivation of Fast and Safe Hypervisor Drivers from Guest OS Drivers. In *ASPLOS*, 2009.